MASTER'S THESIS

A flexible framework for leveraging verification tools to enhance the verification technologies available for policy enforcement

Larkin, James

Award date:
2009

A Flexible Framework for Leveraging Verification Tools to Enhance the Verification Technologies Available for Policy Enforcement

James Larkin

A Thesis submitted for the degree of Master of Science (Computer Science)

School of Information Technology

Bond University

March 31, 2009

Statement of Originality

The material in this thesis has not been previously submitted for a degree or diploma in any university. To the best of my knowledge this thesis contains no material previously published or written by another person except where due acknowledgment is made in the thesis itself.

James Larkin

Acknowledgments

Firstly and foremostly I would like to thank my supervisors Dr. Padmanabhan Krishnan and Dr. Phil Stocks for their time, feedback and their continual support over the years of my research. I would especially like to thank them for their efforts and for proof-reading drafts of this thesis.

I would also like to thank the Australian Government for providing me with an Australian Postgraduate Research Award.

I would also like to thank my family for their on going support. In particular the vital role of my parents for their continual love and encouragement and my brother Henry for his feedback on drafts of this thesis and continual encouragement.

In addition, I would like to thank my office mates Shane Bracher, Percy Pari Salas, James Montgomery and Keith Hales who over the years made my time very enjoyable and were always there for support, encouragement and feedback.

Finally, I would like to thank all my friends, in particular Adrian Gepp, for their support and for making the duration of my thesis very enjoyable.

Abstract

Program verification is vital as more and more users are creating, downloading and executing foreign computer programs. Software verification tools provide a means for determining if a program adheres to a user's security requirements, or security policy. There are many verification tools that exist for checking different types of policies on different types of programs. Currently however, there is no verification tool capable of determining if all types of programs satisfy all types of policies.

This thesis describes a framework for supporting multiple verification tools to determine program satisfaction. A user's security requirements are represented at multiple levels of abstraction as Intermediate Execution Environments. Using a sequence of configurations, a user's security requirements are transformed from the abstract level to the tool level, possibly for multiple verification tools. Using a number of case studies, the validity of the framework is shown.

Contents

List of Figures

Chapter 1

Introduction

The use of software programs is increasing every day. Software programs are produced by sources ranging from known companies to unknown entities and the end-user who also executes the program. Ensuring that these programs are *secure* and do not harm users' computers is increasingly important.

Consider web pages as an example of a security risk. More and more users are viewing web pages that contain embedded JavaScript. These embedded Javascript programs might, for example, upload a number of files specified by the user to a designated server. Without any security measures, the user has no guarantee that the JavaScript program uploads only the specified files to only the designated server. All the user's security requirements, such as a guarantee that only the specified files are uploaded, can be stated in a *policy* that all programs must adhere to prior to being executed.

Potentially *unsecure* or *untrusted* programs can be received in a variety of ways including

- Viewing web pages as mentioned above,
- Opening emailed electronic documents with embedded code. For example, code can be embedded within a PostScript file such that it is executed when viewed with Ghostscript; and,

- Adding external hardware, such as USB memory sticks, to existing hardware.

Regardless of their source, ensuring the security of software programs is essential.

1.1 Program Verification

What exactly constitutes program security varies from user to user. However, each user has requirements that programs must adhere to for them to be allowed to be executed on their machine. For some systems, such as Microsoft Vista, Mac OS and the Java VM, there is an inbuilt access control mechanism that allows a specified level of security to be enforced within the system.

There are two main ways to ensure that a program behaves properly. The first is to restrict the types of programs that can be executed by revoking certain privileges. These privileges can be revoked by the language that the program is implemented in, such as sandbox environments, or by filtering system calls to deny access to potentially dangerous systems calls. The main problem with such mechanisms is that they limit the expressiveness of executable applications.

The second main way is to check or *verify* a program. *Program verification* is the process of determining whether a program adheres to a user's security requirements specified in a *policy*. Manually checking this requires a user to look through and analyse software code to determine if the requirements are satisfied. In practice, this is unrealistic for even small programs because, even by an expert, it is extremely difficult and prone to human error. Therefore, tools that automate program verification are necessary.

Automated program verification can be performed by one of two main techniques. The first is to monitor the execution of a program, that is, dynamic program verification performed at runtime. Note that attacks based on buffer overflows [CPM+98] can be prevented by ensuring appropriate run time checks. Checking safety at run time however

could have a significant speed penalty. The second technique is to statically check a program before it is executed. Approaches like *model checking* [CEE+01] and *theorem proving* [Bib87] can be used to statically verify a program.

Model checking is in principle an automatic technique that can be used to ensure that a program possesses some required properties. Model checking requires the program to be modeled as an automaton with its properties expressed in a form of logic. For model checking to be automatic a model that represents the program as a finite state automaton is required. *Theorem proving*, on the other hand, is a semi-automatic technique that relies on the program and its properties being precisely specified in a form of logic. While it is possible to automate a large proportion of theorem proving, sometimes an expert is required to guide the system. Combining static analysis with dynamic checking has also shown promise. Huang et al. [HYH+04], for example, show how security of web applications can be improved by using a combination of static analysis and runtime monitoring.

1.2 Limitations of Program Verifiers

No verification tool can determine satisfaction of all user's security requirements on every program. There are three main reasons for this:

1. The input language of the program,
2. The method used to verify the program, and
3. The type of security requirements that need to be verified.

There are many options for the input language of a program including source code in one of the many programming languages that currently exist (e.g., Java, C and Haskell) or compiled code. As a result of the numerous options, no current verification tool can

handle all program input languages. In fact, most verification tools accept only a single program input language.

Furthermore, the method used to perform verification determines the policies that can be verified. For example, tools that implement theorem proving techniques are capable of verifying all policies specified in a form of logic, but specifying programs in a form that can be verified by a theorem prover is not easy. On the other hand, an abstraction of a model appropriate for a model checking tool can be created relatively easily; however, model checking tools have a limited set of policies that can be verified on given models.

Using a single tool for program verification is sometimes insufficient. To verify all of a user's requirements of a given program, multiple verification tools may be required. The motivation for this dissertation is to contribute to the problem of how best to utilise and combine multiple verification tools.

1.3 Goals of this Dissertation

Using multiple verification tools to verify a program typically requires multiple security policies to be specified, one for each verification tool used. The main disadvantage of this process is that there is an overhead cost associated with specifying multiple security policies. However, by combining policy languages, a user's security requirements can be represented once in a single language and used by one or more verification tools to enforce those requirements on a given program.

To date, limited work has been done on combining policy languages. The aim of this dissertation is to address a part of this research gap by describing a framework for leveraging verification tools to enhance the verification technologies available for policy enforcement. Specifically, this dissertation makes the following contributions:

1. Using the concepts viewed in existing policy languages, language theory is used to

capture a user's security requirements for multiple representations.

2. Using language theory, a framework is built for transforming a user's security requirements. The framework allows a user's security requirements to be automatically transformed into multiple representations for multiple verification tools.

3. To aid in the transformation process, manually created *configurations* that describe how to transform between representations and how to link these transformed representations are included. Appropriate configurations, such as these, allow a user's requirements to be more easily updated and maintained.

4. What it means for a program to satisfy a user's requirements in terms of the generated representations of those requirements and the tools used to enforce those requirements is clearly defined.

These specific contributions advance current understanding of the best way to combine policy languages in an effort to allow multiple verification tools to be used without the overhead cost of specifying multiple security policies.

1.4 Structure of this Dissertation

The remainder of this dissertation is structured as follows. Background material describing policy representations, verification tools and their current limitations is presented in Chapter 2. Chapter 3 describes both the language for representing a user's requirements and the framework for transforming between these representations. Chapter 4 then validates the framework via case studies and shows the flexibility of the framework by using configurations. Finally, Chapter 5 concludes the dissertation with a discussion of the framework and possible future research.

Chapter 2

Literature Review

Program verification can be divided into two parts: representation and evaluation. Representation deals with representing a user's security requirements in a suitable language, while evaluation deals with determining if a program adheres to the user's security requirements.

This chapter is organised into three sections. Section 2.1 describes the current work on representing a user's security requirements. Section 2.2 describes the current techniques for determining program verification. Finally, section 2.3 describes the main framework that exists for combining policy languages.

2.1 Policy Specification

A security policy consists of sets of constraints on a system's behaviour [BS03]. A security policy is a syntactic representation of the security properties (or requirements) a user wants a program to satisfy. Security properties specify the security goals to be achieved and the language used to encode a security policy depends on the security properties that must be enforced. Security properties can be divided into four main categories [VPS02, BS03]:

1. Confidentiality,

2. Authenticity,

3. Integrity, and

4. Availability.

The most popular security property is *confidentiality* [BS03]. Confidentiality deals with the protection of confidential information and ensuring access to resources is restricted to only to users with authorised access. When a request for access to a resource is made, the system needs to know from whom the request came in order to determine if the request should be granted or denied. *Authentication* deals with confirming a user's identity. Integrity deals with data modification and ensuring that data is complete through transfers, storage and revival. Finally, availability deals with the interruption of system functionality and the ability to access system resources. That is, ensuring that if a user has authorised access to a given resource then they are not prevented from accessing that resource. Traditionally, availability has been the security property that has received the least attention from the scientific and academic community [Sta02].

The remainder of this section discusses how these properties can be represented. Section 2.1.1 describes three main models for representing requirements while section 2.1.2 describes security policy representations for representing the four main categories of policies.

2.1.1 Models of Security

Three models of security are described in this section: the Bell-LaPadula model, the Biba integrity model and the Chinese Wall model.

The Bell-LaPadula Model [Bis03] is a way of modeling confidentiality requirements. In

the Bell-LaPadula Model, subjects (users) and resources (objects) have *security classifications* that specify the level of security of each user and resource. Security levels range from the most sensitive (e.g. "Top Secret") down to the least sensitive (e.g. "Unclassified" or "Public").

The Bell-LaPadula Model combines mandatory and discretionary access control with two distinct Mandatory Access Control (MAC) properties.

1. The *Simple Security Property* states that a subject at a given security level may not read any object at a higher security level.

2. The $^* -$ *Property* states that a subject at a given security level may not write to any object at a lower security level.

These two properties are characterised by the phase *no read up, no write down* which ensures that secure information is not leaked.

The Biba integrity model [Bis03] describes a set of access control rules designed to ensure data integrity. In the Biba integrity model, users and resources are grouped into ordered levels of integrity. Integrity levels differ from security levels in that the higher the level of integrity, the more confidence one has that a program will execute correctly. Thus, data and information at a high level is more accurate.

Similar to the Bell-LaPadula model, the Biba integrity model defines two MAC properties:

1. The *Simple Integrity Axiom* states that a subject at a given integrity level may not read an object at a lower integrity level.

2. The $^* -$ *Integrity Axiom* states that a subject at a given integrity level must not write to any object at a higher level of integrity.

These two properties are characterised by the phase *no read down, no write up* which ensures the integrity of data.

The Chinese Wall model [Bis03] is a model of security that refers equally to confidentiality and integrity. A Chinese Wall is an informal barrier to separate and isolate users to avoid 'conflict of interest' problems. The basis of the Chinese Wall model is that users are only allowed access to information which does not conflict with any other information to which they already have access. Thus unlike Bell-LaPadula, access to information is not restricted by attributes of the information, but by the information for which a user already holds access rights.

In the Chinese Wall model, information is grouped into related classes. In the business world for example, this would typically be information concerning the same corporation. The Chinese Wall model specifies that access is granted if and only if the information is in the same class (that is, within the confines of the wall), or belongs to a different class that does not have a 'conflict of interest' issue. Referring again to the business world, competing corporations would be in different classes that were in conflict with each other so competing corporations could not access information on one another. A further restriction is that if a user requests write access within a class then its read access is strictly restricted to within that same class.

2.1.2 Security Policy Representations

The previous section describes three models for reasoning about security. There are many ways these models can be implemented to specify a user's security requirements. This section describes the main mechanisms for representing policies that specify a user's security requirements.

Access Control

Access control is one of the most widely used security mechanisms [HMHX07], which is typically used for specifying confidentiality and integrity requirements. The simplest way to define users' control to resources is by using an access control list (ACL). ACLs are lists of permissions attached to resources that state who or what is allowed to access each resource and what operations are allowed to be performed on that resource.

One of the simplest implementations of an ACL is an access matrix [Lam74]. Access matrices are simple tables stating the rights users have on resources. An example is below.

	File1	File2	File3
Alice	{read,write}	{read}	{execute}
Bob	{read,write}	{read}	{execute}

While access matrices are simple and work well in centralised systems, in the real world where systems are decentralised and users' access rights are frequently changing, they do not suffice. This is because traditional ACL systems assign permissions to individual users which becomes cumbersome in a large system.

An alternative to ACLs is role based access control (RBAC), in which roles are created and permissions to perform operations are assigned to specific roles. RBAC also differs from ACLs in that ACLs assign permissions to specific files whereas RBAC assigns permissions to operations. For example, an ACL could be used to allow a user read access to a certain file, but this access does not specify how that file could be changed. Using RBAC, a user in a particular role may be allowed to perform a certain operation that, for example, populates a file with records.

Security Enhanced Linux, SELinux [LS01a, SF01, SVS01, Sma02, LS01b], extends Linux with a flexible role based access control mechanism. The access control mechanism allows SELinux to enforce an administratively-defined policy over all processes and users

in the system. Security policies for SELinux are encoded as access vector rules specifying the permissions that users and groups have on objects.

KAoS [BDC+95], Rei [Kag02, KFJ03] and Ponder [DDEL01] are all policy languages for representing access control requirements. Each language allows four types of policies to be encoded: positive and negative authorisations and positive and negative obligations. Positive and negative authorisations refer to the rights a user has and does not have respectively. That is, authorisation requirements specify the actions a user is and is not allowed to perform. Obligation requirements on the other hand refer to the actions that *must* be performed and are typically event driven. Thus, when a specific event occurs, some action must or must not be performed in the case of a positive or negative obligation respectively. Each language then represents these types of policies in a different syntax.

An alternative approach to access control taken by Venkatakrishan et al. in [VPS02] is to empower programs rather than typically disabling them. Venkatakrishan et al. represent policies as Extended Finite State Automata (EFSA), which allows the enforcement of these policies to be made efficiently via runtime monitoring. An EFSA has states and transitions on pairs of states like conventional finite state automata. In addition, EFSAs are augmented with variables along the transitions which are used to store event arguments.

In Venkatakrishan et al.'s work, the security-relevant behaviour of a program is modeled in terms of sequences of externally observable actions, or events. In the context of Java for example, such events include method entries and exists. A security policy then specifies constraints on the sequence of such events that may be produced by a program. These constraints are implemented using the transition variables in EFSAs.

While access control policies are primarily used for encoding confidentiality requirements, they can also be used to represent integrity requirements. As an example, Jajodia et al. in [JSS97] show how access control policies can be used to represent separation of duty requirements. To achieve this Jajodia et al. state policies as mappings from 4-tuples

(object, subject, role set, action) to either authorised or denied.

Access control policies have also been extended to include availability properties. Evans et al. in [ET99] show how simple access control policies can be extended to address resource usage. Evans et al. use *resource descriptions* to abstractly describe system resources and the ways they can be manipulated. In this case, policies are defined by attaching code checks to resource operations.

Information Flow

Information flow is a different mechanism for specifying confidentiality requirements. Information flow control deals with controlling the flow of information to ensure confidential information remains confidential and is not leaked. Much work on information flow has been done by Myers et al. [ML97, ML98, Mye99, ML00, SM03, CM04]. In general, the secrecy of data and controlling the flow of information can be represented by putting resources and users into classes. Different classes define different levels of security. Ensuring the confidentiality of information is then ensuring that users in one class cannot access resources in another of a higher security level (as in the Bell-LaPadula Model).

One common method for performing this is by defining security classes as types and using a set of typing rules to control the flow of information between those classes. There are many type systems for this, most of which are based on the spi calculus [AG97]. Abadi's type system [Aba99], Abadi and Blanchet's type system [AB01] and Gordon and Jeffrey's type system [GJ02] are examples of type systems for specifying information flow policies.

Abadi and Blanchet's type system [AB01] is briefly described here as an example of how types can be used to encode information flow control policies. In their system they define two main types: public and secret. Public is the type given to all data that is public, that is, data that can be accessed by anyone. Secret is the type given to all data that is secret, that is, data that cannot be accessed by an attacker. All data, channels and encryption

keys are given a type. Type checking then ensures that no data of type secret is accessed by something with type public. As an example, let us consider a public channel b, which is used to output two pieces of information: public data, and a public channel that itself is used to send secret data. In this example, b would have the type

$$b : C^{Public}[Public, C^{Public}[Secret]]$$

.

Similar systems can be used to specify integrity policies. Zdancewic in [Zda03] describes a type system using various levels of security lattices. Here, security lattices are similar to types, however, they describe the integrity level of data rather than the confidentiality level of data.

Access Control Using Logic

One problem with many policy languages is that they are tied to their implementation. Policy languages that are encoded in logic move away from the implementation and separate policy from mechanism.

In 1992, Woo and Lam [WL92, WL93] created a logic for representing authorisation requirements. Their logic was based on representing three structural properties apparent in authorisation requirements: closure, default and inheritance properties.

Closure properties are used to ensure the consistency of authorisation requirements. For example, if a user is allowed write access to a file, they should also be allowed read access. Default properties are used to define the course of action that should be taken in the absence of any applicable explicit policies. Most real systems use defaults properties. The most common example is a *restrictive* policy whereby access is denied unless explicitly authorised. Inheritance properties are used to precisely define how to resolve any inconsistencies in a policy. For example, if a user belongs to two groups where one group allows

read access to a file and the other group denies read access to the same file, what access should the user have? Inheritance properties answer this question.

Given these three structural properties, Woo and Lam devised a logic for representing authorisation requirements. Authorisation requirements in their logic are encoded as rules. A rule has three components: a prerequisite (f), an assumption (f'), and a consequent (g), written $\frac{f:f'}{g}$. The meaning of a rule is that g is enforced if f is true, and there is no other rule contradicting f'.

Given a set of rules specifying a user's requirements, before a user can perform a particular access on a resource, the user must first obtain the access rights for that resource. This is done by submitting a request to the *authorisation module* which is responsible for determining which requests to grant or deny. The authorisation module uses a set of rules in the logic to make this decision.

A similar logic is described by the previously mentioned Jajodia et al. in [JSS97]. Jajodia et al. define an Authorisation Specification Language (ASL) that maps 4-tuples (object, user, role set, action) to either authorised or denied.

Lampson et al. in [ABLP92, LABW92] describe a theory of authentication in distributed systems that allows the source of a request to be identified. Their theory contains access control mechanisms as well as a notion of principals and a "speaks for" relationship between those principals to reason about their authority. A principal's authority can be deduced by the principals that it can speak for.

Trust Management Systems

Trust management systems introduced by Blaze et al. [BFL96] are a unified approach to specifying and interpreting security policies and credentials that describe a specific delegation of trust. Trust management systems unify credentials, access control and authorisations. Security policies and credentials are defined in terms of predicates, called

filters, which are associated with public keys. Each user is represented by a public key. Filters accept or reject actions based on the rights a user is trusted to perform.

There are a number of example implementations of trust management systems including PolicyMaker [BFS98], Keynote [BFK99], REFEREE [CFL+97], Delegation Logic [Li00], SD3 [Jim01] and Protune [BO05].

2.2 Policy Enforcement

This section provides an overview of some of the current verification technologies available for ensuring a program adheres to a user's security policy. Section 2.2.1 introduces the main attributes for characterising verification technologies with the rest of the sections outlining verification technologies that exist today. Section 2.2.4 gives a brief summary of the verification technologies discussed in this section and outlines an issue of concern that this dissertation aims to address.

2.2.1 Characterising Security Policy Verification Technologies

There are many verification tools that exist for determining if a program satisfies a security policy. Each verification tool implements a verification technology. There are many verification technologies that currently exist, each of which differs from another by four main factors:

1. Work distribution between a user and a foreign source,
2. Manual vs. automatic verification,
3. Syntactic vs. semantic verification, and
4. Static vs. dynamic verification.

Verification relies on the generation of a model, abstraction or proof. Work distribution refers to whom performs the generation and verification of these models, abstractions and proofs. This is of main interest in mobile program scenarios where a program is created at a different location from where it will be executed. Work distribution also refers to the amount of work performed at both the creation and execution locations. In the case where users create and execute their own applications, they also perform all the work for verification.

Verification can be performed automatically or manually. Automatic verification of a program has the advantage of not requiring human interaction, however, it lacks the ability to perform complete verification as some properties may be undecidable or intractable.

Verification can be performed using the syntax or semantics of a program. Syntactic verification technologies perform verification based on the syntax of a program while semantic verification technologies perform verification based on what the program does.

Finally, verification can be performed statically, before a program is executed, or dynamically, while the program is being executed. Statically checking a program has the advantage of ensuring that a program is secure prior to its execution, however, it lacks the ability to verify dynamic information such as user input. Dynamically checking a program allows for dynamic properties to be checked (e.g., user input). It is possible when dynamic checking is performed via program monitoring that an unwanted and unsatisfactory operation may occur before it is checked, and undoing the effects of a partially run program (e.g., files partially updated) is a major issue. Furthermore, dynamic verification has a significant speed penalty as checks must be performed while the program is being executed. In contrast, static checking has the advantage of verifying once and then executing multiple times.

2.2.2 Restriction Mechanisms

One method for ensuring program security is by using a suitable mechanism to restrict the types of programs that can be executed.

Key Infrastructures

Key infrastructures are used to ensure trust properties. Trust is the extent to which one party is willing to depend on another [PS05]. Key infrastructures are a technology where a user will execute a program only if it's from a particular user whom they trust. There are a number of key infrastructure mechanisms including Public Key Infrastructures (PKI) [KHPC01], Simple Public Key Infrastructures (SPKI) [CEE+01, Eli98, HK00] and Simple Distributed Security Infrastructures [RL96].

While these technologies are widely used and allow a user to know from whom a program came, they say nothing about the security of the actual program which may indeed be faulty, intentionally or not.

Sandbox Environments

Sandbox environments are virtual zones in which untrusted programs are executed. The virtual zones are restricted zones allowing only a restricted set of operations to be executed on a restricted set of resources. Sandbox environments can be implemented in one of two ways or in a combination of both. The syntax of the programming language used to encode the program can be designed to restrict certain operations such that only a safe set of programs can be encoded within the language. The second approach is where programs are executed within a virtual environment such that they only have access to virtual memory and virtual resources. The virtual environment allows no access to resource outside the environment ensuring the system's resources are secure from any program executed within the virtual environment.

One of the most popular examples of a sandbox environment is the Java Sandbox [Gon98, GMPS97]. The Java Sandbox model implements both of these features by restricting the language to ensure it is dynamically type-safe and using the Java Virtual Machine to mediate crucial system calls to ensure in advance that they are secure.

While the Java Sandbox provides a safe environment for programs to be run, it restricts the types of programs that can be executed. BlueBoX [CC03] is an intrusion detection system for controlling a process's access to system resources. BlueBoX lessens the restrictions of a hard-bound sandbox environment, such as the Java Sandbox, by allowing a user to define the bounds of the sandbox by defining the allowable system calls of programs. The major drawback with BlueBoX is that the kernel needs to be patched to include the *enforcer* which allows and disallows system calls based on specified policy rules.

2.2.3 Verification Checking Mechanisms

Another method for ensuring program security is by checking that the program adheres to a user's security requirements. This section describes technologies for checking a program.

Model Checking

Model checking [CGP99] is an automatic technique for verifying correctness properties of a program. The model checking procedure performs an exhaustive search of the bounded/finite state space of a program to determine if some specific properties hold. The model checking procedure requires two inputs: a finite-state model which represents the program being verified and a specification stating the properties that the program must satisfy. One of the major benefits of model checking is once the model and policy specification are specified, verification is fully automatic and performed before a program is executed. Furthermore, given the unsuccessful verification of a program, model checking techniques show, via an error trace, why a program failed verification.

As stated, model checking depends on the behaviour of a program being represented as a finite-state model. One problem with this is correctly representing a program as a finite-state model. This is known as the model construction problem [CGP99]. Another problem with constructing this model is the state-explosion problem, where, even for small programs, the number of states of a program is often far greater than can be handled by a standard computer or often even a supercomputer [Val98]. By using suitable abstractions [BMMR01, BR02, BB02, CCG$^+$03, FQ02, HJMS03], large programs with possibly infinite state can be represented. The downside to abstracting a model from a program is that during the abstraction process information from the program is lost. For example, consider a program that includes a system call to print a line of text to a given printer. The abstraction of this program might not differentiate this system call with printing a line to another printer or to the screen.

The behavioural properties that a program must satisfy (specification) are usually expressed in some form of temporal logic [HR04]. Common temporal logics used are Linear Temporal Logic (LTL) and Computational Tree Logic (CTL).

Linear Temporal Logic

Linear Temporal Logic (LTL) is a mechanism for reasoning about properties over time. LTL is used to encode formulae about the future of paths in a computation tree where a computation tree represents all possible behaviours of a system.

LTL formulae are defined over atomic propositions. The set AP is defined to be the set of all atomic propositions and it is assumed there exists the function $L : S \rightarrow \mathbb{P}(AP)$ that labels each state with a set of atomic propositions that are true in that state. The syntax for an LTL formula is below. The syntax includes the temporal operators always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$) and until ($\mathbf{U}$).

$$
\begin{aligned}
LTL \;::=\; & a \in AP \\
& \mid \neg LTL \\
& \mid LTL \wedge LTL \\
& \mid \Box LTL \\
& \mid \Diamond LTL \\
& \mid \bigcirc LTL \\
& \mid LTL \;\mathbf{U}\; LTL
\end{aligned}
$$

The semantics of an LTL formula is given below where w is a sequence of states and i is the current state.

$$
\begin{aligned}
w, i \models p \quad & \Leftrightarrow p \in L(w(i)) \\
w, i \models \neg\varphi \quad & \Leftrightarrow w, i \not\models \varphi \\
w, i \models \varphi \wedge \phi \quad & \Leftrightarrow w, i \models \varphi \wedge w, i \models \phi \\
w, i \models \Box\varphi \quad & \Leftrightarrow \forall\, n \geq i \bullet w, n \models \varphi \\
w, i \models \Diamond\varphi \quad & \Leftrightarrow \exists\, n \geq i \bullet w, n \models \varphi \\
w, i \models \bigcirc\varphi \quad & \Leftrightarrow w, i+1 \models \varphi \\
w, i \models \varphi \;\mathbf{U}\; \phi \quad & \Leftrightarrow \exists j \geq i \bullet w, j \models \phi \wedge \forall\, n \mid j > n \geq i \bullet w, n \models \varphi
\end{aligned}
$$

Model Checking Tools for Programming Languages

Popular model checking tools include SPIN [Hol97], dSPIN [DIS99], Java PathFinder [HP00], Bandera [CDH+00], Blast [BHJM07], Magic [CCG+03], MOPS [CW02] and SLAM [BR02].

One of the most popular is SPIN [Hol97]. SPIN accepts programs in the PROMELA language and correctness properties in LTL. SPIN automatically translates a LTL formula into a Büchi automaton [GEP+95] and a program encoded in the PROMELA language into a finite state automaton. To perform verification, SPIN calculates the synchronous

product [Hol97] of the Büchi automaton and the finite state automaton. The synchronous product of two automata is the intersection of the two. The resulting Büchi automaton specifies the program behaviours that satisfy the LTL formula. If the resulting automaton is empty then there is no behaviour of the program that satisfies the LTL formula.

SPIN can only verify standard safety properties and a limited range of liveness properties [Hol97]. Further, SPIN relies on the program to be verified being represented in the PROMELA language. Java PathFinder [HP00, VHBP00] and Bandera [CDH+00] are both tools which take a Java source program and translate the program into the PROMELA language for the SPIN model checker.

Not all tools use an underlying model checker like SPIN. Blast and Magic for example are both model checkers for checking C programs that use their own model checker.

Blast [BHJM07, HJMS03] is a verification system for checking safety properties of C programs. Similar to the process in Java PathFinder (JPF), safety properties in Blast are specified using *assert* statements throughout C source code. Given a C program instrumented with assert statements, Blast transforms this into a suitable model and performs a reachability analysis on the model to perform verification.

Magic [CCG+03] is a model checking tool that is capable of automatically checking conformance of a C program against a finite state machine. Unlike Blast and JPF, the Magic framework uses Labeled Transition Systems (LTS)s to express specifications. Verification is performed by firstly extracting a model of the program using a technique called *procedure abstraction*, and then this model is checked against the LTS specification.

Theorem Proving

Theorem proving is a semi-automatable verification technique for proving program properties in a mathematical sense. Theorem proving relies on the program being specified clearly and precisely in a form of logic. Typically, classical first-order logic is used, how-

ever, higher-order logic is sometimes used. The underlying strength of theorem proving is that the program and problem to be verified are precisely stated without ambiguity. However, encoding this in a suitable logic requires an expert.

Unlike model checking, theorem proving techniques can either prove or disprove a theorem. They cannot give an error trace like model checkers can showing how verification failed. As a result of the complexity of some proofs, theorem proving sometimes requires an expert to guide the system.

Theorem Proving Tools for Programming Languages

Currently there are many implementations of theorem proving tools (theorem provers) including Elf [Pfe89], Twelf [PS99], λProlog [FGMP90, NM88], LogScheme [RW89], and Prototype Verification System (PVS) [OSR92]. Some of these use the Edinburgh Logical Framework (LF) [AHM89, Pfe91, Pfe96] as the underlying logic. Other theorem provers use alternative means to provide the same functionality. These examples are just a few of the theorem provers that exist.

Elf is a tool that supports a metalanguage for proof manipulation [Pfe89]. It is intended for meta-programs such as theorem provers. The metalanguage unifies logic definition with logic programming (LF with Prolog). It achieves unification by giving types an operational interpretation the same way Prolog gives clauses an operational interpretation. The metalanguage is a strongly typed language, since it is directly based on LF.

Twelf is also a tool that supports a metalanguage for specifying, implementing and proving properties of programming languages and logics [PS99]. Twelf's metalanguage is an implementation of LF. Twelf is the successor of Elf, thus is much the same as Elf in syntax and semantics. Twelf's metalanguage employs 'constraint simplification' and carries along equational constraints in a normal form. This is used instead of unification because LF is undecidable. Constraint simplification involves two equivalent constraints representing

the same information, but, one may be simpler than the other. The implementation of Twelf comprises three major parts. The first is the core type theory which provides the infrastructure for representing specifications, algorithms and meta-theory. Algorithms are executed by the constraint logic programming engine, which is the second part of the Twelf system. The final part is the meta-theorem proving component that supports the meta-theory.

λProlog is a logic programming language that extends Prolog by incorporating notions of higher-order functions, λ-terms, higher-order unification and polymorphic types [FGMP90]. The reason for considering this extension is for the use of representing proofs and theorems. λProlog is very similar to Elf and Twelf though it differs in the sense that λProlog doesn't support LF. Instead, λProlog provides these new features by extending the classical first-order theory of Horn clauses to the higher-order theory of hereditary Harrop formulas [FGMP90].

LogScheme is an experimental language where the main features of logic programming, nondeterminism and unification are added into the functional programming language Scheme [RW89]. LogScheme differs from other implementations of theorem provers in that these new features are added into a functional programming language, rather than incorporating functional programming language terms into a logic programming language.

PVS is verification system, that is, a specification language integrated with tool support and a theorem prover. The specification language is based on classical, typed higher-order logic. The theorem prover provides a collection of primitive inference procedures that are applied interactively under user guidance.

Proof-Carrying Code

Proof-Carrying Code [Nec97, NL96, NL98] (PCC) is an architecture for ensuring the safety of mobile programs, where mobile programs are those created at one location by a code

producer and executed at another location. In the PCC architecture, code producers create proofs of their programs that prove they adhere to a user's security policy. A producer sends this safety proof along with the program to the user. Upon receiving a safety proof and program, a user performs two stages of verification. First, the user checks to see if the supplied proof adheres to the user's security policy. Secondly, the user checks the proof against the program to ensure it has not been falsified.

PCC uses theorem proving techniques to prove the validity of a program, however, the architecture distributes the work load by placing the burden of generating a proof on a code producer. This greatly reduces the work load of recipients since checking a proof is much easier than generating a proof. Further, since a producer creates the proof, this one proof can be sent to all recipients of the application. However, since the producer is creating the proof, an extra verification stage must be performed to ensure the proof generated by a producer has not been falsified.

Current implementations of the PCC architecture use the Edinburgh Logical Framework to encode proofs. Theorem proving is then used to check the proofs and type checking is used to ensure the proof has not been falsified. The PCC architecture has been used to verify both C [Nec97] and Java [CLN00] programs.

Model-Carrying Code

Model-Carrying Code [SRRS01, SVB+03] (MCC) is an architecture similar to PCC and was designed by Sekar et al. as an alternative to the PCC architecture [SVB+03]. In the MCC architecture, a model is used instead of a proof and model checking techniques are used to determine if the model adheres to a user's security policy.

Sekar et al. describe three ways to ensure a generated model has not been falsified [SVB+03]. The first way is to use a monitor at runtime to check the consistency of the model with the program. This would essentially reduce verification to run-time checking. The

Technology	Features of Verification Technology			
	Onus of Proof	Static v Dynamic	Syntax v Semantic	Automatic v Manual
Sandbox Environment	Embedded within language	Dynamic	Syntax	Automatic
Model Checking	User	Static	Semantic	Automatic
Theorem Proving	User	Static	Semantic	Manual
PCC	Producer	Static	Semantic	Manual
MCC	Producer	Static	Semantic	Automatic

Table 2.1: Summary of Verification Technologies

second way is to use trust mechanisms to ensure the sender of the model is trusted. This would ensure the creator of the model is authenticated and trusted but wouldn't ensure the consistency of the model. Finally, the third way is to use the PCC architecture to formally verify the model. While a model would be substantially smaller than a program, the overall work load required to generate and verify a proof of the model would be substantial. Currently, the first method is used.

2.2.4 Verification Technology Summary

Numerous verification technologies exist. Each technology has its abilities and limitations. Table 2.1 gives a summary of the features of each of the technologies described in this chapter. Key technologies are omitted from the table as the table's features are concerned with the analysis of programs for verification.

The key feature apparent in verification technologies is the semantic analysis of a program performed statically. This allows a program to be analyised prior to its execution. Semantic analysis of a program is ideal as verification is based on what a program does and not just on its syntax. For some policies however, dynamic analysis may be required.

While automatic verification allows a user to click "go", manual verification uses an expert to provide information to help the system when verifying more complex policies.

Therefore, the option of whether to use manual or automatic verification depends on the policies being verified as well as who the user is, i.e., an expert or non-expert.

Each user has a set of requirements which specifies what it means for any program executed on their machine to be secure. Given a user's security policy, there may not always exist a single tool that is capable of determining if all program behaviours of any program adhere to that security policy. To determine satisfaction of a policy, multiple tools may be required. Using multiple tools from different verification technologies allows a greater range of security policies to be verified on a greater range of program behaviours.

A problem with using multiple verification tools is the requirement of multiple representations of a user's requirements: one for each tool used to perform verification. By combining policy languages, multiple tools' representations can be represented as a single representation. The next section describes current work on combining policy languages.

2.3 Analysis of Techniques for Leveraging Verification Technologies

Each verification technology has its own niche and performs well in its own domain. While each technology can be improved within its own domain, the work presented in this dissertation focuses on combining languages and technologies to cover more application domains.

Using multiple verification technologies can enable more application domains to be verified. Supporting multiple policy languages can allow multiple verification technologies to be used to verify an application.

2.3.1 The Heterogeneous Tool Set

There has been limited work done on combining policy languages. Mossakowski et al. [Mos05, MML07] developed the Heterogeneous Tool Set (Hets), which is a tool set for combining specifications among different logics using logic translations. Hets was created with the realisation that large software systems are typically best specified using a number of specification languages. Common examples include logics for specifications of datatypes, logics for reasoning about space and time, and logics for specifying security requirements and policies. Hets allows a system to be specified in a number of specification languages while still allowing the system as a whole to be verified using one or more proof checkers. To allow different provers to be used, proof goals can be *translated* through logics until one with proof support is reached.

The Hets framework allows a system to be specified using multiple logics and allows different parts of a system to be proved using one or more different provers. In Hets, a system is specified as a *heterogeneous specification* that allows multiple specifications to be linked together. Heterogeneous specifications are based on individual logics and logic translations. A logic in Hets has a notion of a signature, and sentences and models constructed over that signature. Sentences specify the syntax of a logic system. That is, all the ways in which symbols in a signature can be constructed together while models define the interpretations of sentences, i.e., the semantics of a logic system. The essential element of a logic is the essence of *satisfaction* that relates sentences (syntax) with models (semantics). The implementation of a logic in Hets consists mainly of syntactic entities for representing signatures and sentences. The current implementation represents the Common Algebraic Specification Language (CASL) in approximately 11,500 lines of code. Each logic used in Hets must also provide a parser that parses an input string for that particular logic.

It is important to know when sentences in one logic can be represented in another logic such that soundness is preserved. That is, sentences that hold in a model in one logic

also hold in the models in the translated logic. *Logic translations* are mappings between sentences in two logic systems such that soundness is preserved. The founding desire for logic translations is to permit a theorem prover for one logic to be used on theories from another logic.

Heterogeneous specifications are used to specify a system in Hets. A heterogeneous specification, formalised as a development graph [MAH06], consists of a set of nodes and a set of arrows. Nodes are specifications that follow the syntax of a logic system while arrows define the links between specifications. Arrows are either *definition links*, that allow axioms of other nodes to be inherited, or *theorem links*, which represent proof obligations between specifications. Proof obligations represent the proof goals that must be proved for the system to be sound.

Proof obligations can be proved by using a local theorem prover if one is available for the current logic. If a prover is not available for the current logic and there exists a sequence of logic translations from the current logic to a logic with a theorem prover attached, then the proof obligations can be translated using the logic translations and proved using that prover.

2.3.2 Conclusion

Combining policy languages allows the capability of using multiple verification technologies to verify a system. The Hets framework allows a system to be specified using multiple specifications with the ability to prove proof obligations between specifications to determine the soundness of a system. The framework relies on each logic being precisely defined along with any logic translations between logics. While in theory each logic must have both a syntax and semantics, their implementation induces the semantics of logics via the theorem provers used. For example, they give Haskell a denotational semantics [TLMM07] based on its translation into Isabelle/HOL which is the input language to the theorem prover

Isabelle.

This dissertation presents a framework for using syntactic based transformations between specifications that alleviates the overhead of semantic definitions. A system in this framework is represented using an automaton which allows a system to be represented at multiple levels of abstraction. This allows a user to specify a system abstractly and use *transformations* to transform that system between various levels of abstraction from the abstract to the concrete (tool) level. Verification tools can then be used to verify the individual tool level representations. Finally, semantics of the abstract system can then be induced from the verification of the tool level representations.

The framework presented in this dissertation allows multiple verification tools to be used to verify a system. Syntactic transformations are used between the representations with the semantics of the system being defined by the verification tools used to verify the system. The framework however relies on the user specifying the transformations to encode transformations that are sound, that is, that the input to the verification tools is appropriate.

The framework presented in this dissertation can be seen as a light-weight implementation of the Hets framework. The Hets framework consists of a graph of defined logics and logic translations. A system is then specified using a heterogeneous specification based on the defined logics. However, in the framework presented in this dissertation, a system is initially specified in some representation. The system represented in some representation is then translated into possibly multiple specifications for particular verification tools. Each specification (apart from the initial one) is induced via the transformation used to produce that specification. This differs from the Hets framework where each logic must be specified.

A system in the Hets framework is sound if all the proof obligations can be proved. Instead of using proof obligations, the framework presented in this dissertation uses a notion of verification. A system in this framework is verified if the verification tools, in conjunction

with the operators used to link specifications, verify the individual specifications. Since a specification in this framework can be transformed into multiple specifications, operators are used to link multiple specifications. It is assumed each operator is suitably defined.

Hets is the main tool available for combining policy languages and as discussed it has both advantages and disadvantages as a tool for combining policy languages. Therefore, there is a need to explore different ways to combine policy languages. This research can lead to finding ways to improve Hets or to providing alternative systems for combining policy languages. The framework presented in this dissertation contributes to the field by providing a light-weight and flexible alternative to Hets that has its own, different, advantages and disadvantages (to be discussed further in later chapters). Thus, it is logical that this framework will be more suitable than Hets in certain situations, such as a situation when being "light-weight" is desirable.

The next chapter describes a framework for leveraging verification tools to enhance the verification technologies available for program verification. This framework allows multiple verification tools to be used to perform verification without the need to manually specify multiple representations of a policy.

Chapter 3

Leveraging Verification Technologies

Each user has a policy specifying what it means for any program executed on their machine to be secure. To accommodate all behaviours of any program, multiple tools may be required to determine program satisfaction. Using multiple tools to verify a given program means a user's security policy requires multiple representations: one representation for each tool used to verify the given program.

There are two ways to create multiple representations of a policy. Method one is to manually create the numerous required representations. One of the problems with this is the effort required if a policy changes, in which case the representations for some, and possibly all of the verification tools need to be changed. Another problem is if a user would like to add a new tool to perform some verification, a new representation for that tool is required. This representation would again need to be manually created. Furthermore, the representation for each verification tool will probably not be a complete representation of all of the user's requirements as each tool probably can not verify all the requirements, otherwise a single tool would have been used. Therefore, it is necessary to keep track of which tool is verifying what properties. Maintaining and updating this record and the representations for each tool is both tedious and error prone.

Method two alternatively involves automating the process of generating representations. The next section introduces a framework for automatically generating multiple representations for multiple verification tools.

3.1 Thesis Framework

This chapter describes a framework for ensuring the security of untrusted programs using multiple existing verification tools. In the framework, a user has an initial representation of the requirements any program must adhere to if it is going to be executed on a user's machine. This representation is represented at a high level of abstraction allowing a policy to be specified independently of specific program behaviours. This representation, along with the untrusted program, is automatically transformed into one tool level representation for each of the one or more verification tools. Each representation is then used by a specific verification tool to verify the given program. Figure 3.1 displays a pictorial view of the framework.

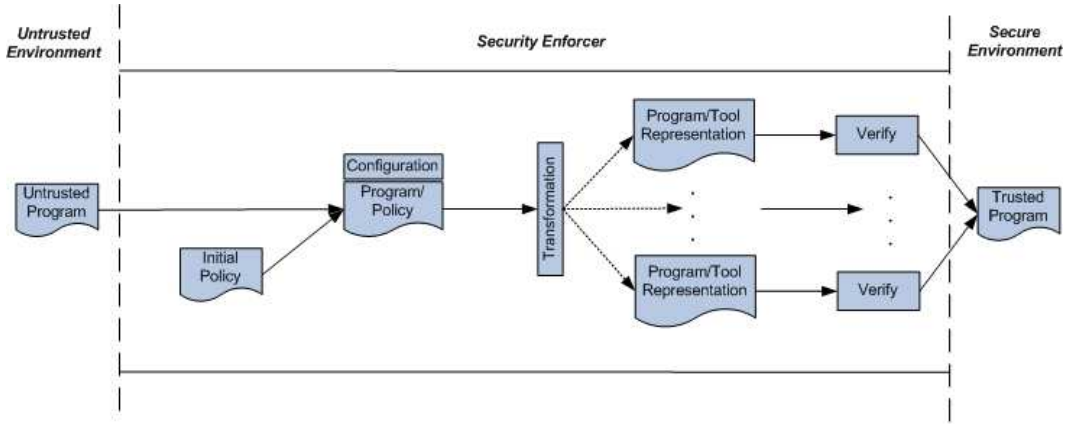

Figure 3.1: Framework

A transformation takes a policy and a program and uses a *configuration* to generate one representation for each of the one or more verification tools, where each representation contains some or all of a user's security requirements. Given the policy, program and

appropriate configurations by a person, the process of generating the representations can be automated. Thus, configurations are manually created to guide the transformation process by specifying the mappings from one policy representation to another.

Given a policy that has multiple representations where each is in the syntax for a specific verification tool, a program is verified by each verification tool using the corresponding representation. Given the successful verification by one tool and the unsuccessful verification by another, what does this mean in terms of program satisfaction of the initial policy? A formal method to produce one overall verification result from one or many individual verification results will be described in the following sections. In brief, when multiple representations are generated, the representations are linked together using a number of operators. These operators are used to specify what overall program satisfaction means in terms of the individual verification results from specific tools. Firstly, the notation used to formally describe the framework will be introduced.

3.2 Notation

A *language*, L, is a possibly infinite set of *strings* over an alphabet, Σ, where $L \subseteq \Sigma^\omega$. The set Σ^ω is the set of all infinite words over the alphabet Σ. An alphabet, Σ, is a subset of the universal alphabet, $\Sigma \subseteq \Sigma^u$. A string is an infinite sequence of symbols from an alphabet. The variable w with possible subscripts is used to represent a string. The notation $w(i)$ is used to indicate the ith symbol of the string w. The first symbol in a string is at position 0. The notation $w[n..]$ is used to indicate the substring of the string w from the nth symbol onwards. The symbol ; is used to denote the concatenation of a symbol onto a string. The variable s with possible subscripts is used to represent a symbol.

The symbol # is used in front of a symbol from an alphabet to denote the occurrence of that symbol. The combination $\#s$ can be used as a predicate.

The set *Val* is used to denote the set of all values. The notation $\bigcup_{u,v \in Val} \{read(u,v)\}$ is used to denote the set of symbols $read(u,v)$ for all values of u and v.

The operator $\bigoplus$ is used to denote the choice of all the regular expressions in a given set. For example, $\bigoplus\{a,b,c\} = a+b+c$. The operator is formally defined in the definition below.

Definition 3.1

$$\bigoplus\ ss = p(ss)$$
where
$$p(ss) = \begin{cases} s + p(lss) & s \in ss \wedge lss \cup \{s\} = ss \\ \varepsilon & \mid ss \mid = 0 \end{cases}$$

The notation $\mathcal{L}(\varphi)$ where φ is a LTL formula is used to specify the language accepted by the LTL formula. The language accepted by an LTL formula is the infinite set of strings that satisfy the formula

$$\mathcal{L}(\varphi) = \{w \mid w, 0 \models \varphi\}$$

where the satisfaction of a given formula is defined in section 2.2.3.

3.3 Execution Environments

To express a user's security requirements, a language can be used to represent those requirements. An Execution Environment (EE) is a language for representing a user's security requirements in the framework.

Security policy enforcement can be seen as restricting the programs that are allowed to be executed. For automata, a program is a set of strings that describe all the possible behaviours of a program. A security policy can be viewed as a filter over the strings in a

program that filters out all the bad behaviours.

An EE is used to represent a user's security policy. An EE specifies a language which defines the allowable strings in a program. A language is defined over a set of symbols from an alphabet. The definition of an EE is given in Definition 3.2. Common languages used to define the allowable strings in a language include ω-regular expressions, Büchi automata and LTL.

Definition 3.2 (Execution Environment)

$$EE : (\Sigma, L)$$

Example *Consider the ω-regular language consisting of only* a*'s and* b*'s, $\Sigma = \{a, b\}$, and a user's policy statement "To be safe, a string must begin with an* a *and consist of only* a*'s and* b*'s." An EE that represents this is*

$$L_{RE} = (\{a, b\}, a(a + b)^{\omega})$$

Example *This example EE uses LTL to specify that if a given user* p *reads to file* r *that user is at no time after allowed to write to that file. The alphabet for the language is the set of all read and write symbols for all values of* p *and* r.

$$L_{LTL} = (\bigcup_{p,r \in Val} \{read(p, r), write(p, r)\}, \mathcal{L}(\bigwedge_{p,r \in Val} \Box(\#read(p, r) \supset \Box(\neg \#write(p, r)))))$$

3.3.1 Transforming and Linking Execution Environments

Execution environments can be transformed from one representation to another. This allows a user to have an initial single abstract policy representation which can be transformed into a specific tool representation. A transformation is a mapping from one EE to another.

Definition 3.3 (EE to EE Map)

$$Map : EE \rightarrow EE$$

Example *This example demonstrates a mapping from one EE to another. The mapping specifies a mapping from a language that allows only sequences of* a*'s to an EE that specifies a language that allows only sequences of* b*'s where each* a *is replaced by a* b*.*

$$f : (\{a\}, a^{\omega}) \rightarrow (\{b\}, b^{\omega})$$
$$f(\{a\}, \rho) = (\{b\}, \{g(w) \mid w \in \rho\})$$
$$\textit{where } g(a;\ w_1) = b;\ g(w_1)$$

To support verification by multiple tools, EEs are linked together via operators. This allows the potential of a single policy to be mapped into multiple representations. An example is depicted in Figure 3.2. The example graph consists of an abstract policy, $IEE_{Abstract}$ and two configurations that map this IEE into two tool specific IEEs, $Tool_1$ and $Tool_2$ linked together via an operator. In the figure, the symbol $\oplus$ is used to indicate an operator linking two representations.

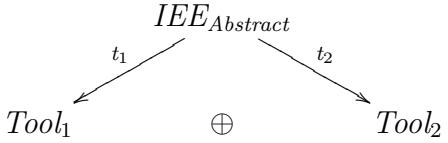

Figure 3.2: An example graph showing the possible configurations to apply to an IEE.

Operators are used to specify the relationships between languages, that is, what satisfaction of the initial policy means in terms of the tool representations of that policy. The are many possible operators that could be included to link languages. This section introduces three operators used in this dissertation: *EITHER*, *BOTH* and *IMP*.

The *EITHER* operator links two EEs specifying that a program must satisfy either EE for that program to satisfy the user's security requirements. The *BOTH* operator links

two EEs specifying that a program must satisfy both EEs for that program to satisfy the user's security requirements. The *IMP* operator links two EEs specifying that if a program satisfies the first EE the program must satisfy the second EE to satisfy the user's security requirements. Three scenarios are now explained to show the usage of each operator.

In scenario one, a user's security requirements are that only one user can write to the file *fileX* at a time. For Java programs, both of the tools Java PathFinder and Bandera can independently be used to verify all of this policy. As long as either Java PathFinder or Bandera verifies that a given program satisfies this policy, then the program adheres to the user's security policy. This is an example of using the *EITHER* operator between two policies.

In scenario two, a user's security requirements are that private information must not be accessed by users with only public access and only one user can write to the file *fileX* at a time. For Java programs, JFlow can be used to verify the information control part of the user's requirements while Bandera can be used to verify that only one user writes to file *fileX* at any one time. For a program to adhere to the user's security requirements, a program must be verified by both JFlow and Bandera. This is an example of using the *BOTH* operator.

In scenario three, the user's security requirements are that if a program uses SQL queries, then the SQL queries should be safe. Here, two tools are needed. The first to check if SQL queries are used, which can be done using syntactic analysis tools. Then, if the first tool verifies that SQL queries are used, another tool is needed to ensure these SQL queries are safe. This is an example of using the *IMP* operator.

The syntax of an EE including these three operators is below. Additional operators can be added as long as an appropriate semantics are given to each added operator.

$$EE : (\Sigma, L) \mid BOTH(EE, EE) \mid EITHER(EE, EE) \mid IMP(EE, EE)$$

The semantics of the operators *BOTH*, *EITHER* and *IMP* are given below.

Definition 3.4 *The semantics of the three operators over an EE is*

$$\begin{aligned} BOTH((\Sigma_1, L_1), (\Sigma_2, L_2)) &= (\Sigma_1 \cup \Sigma_2, L_1 \cap L_2) \\ EITHER((\Sigma_1, L_1), (\Sigma_2, L_2)) &= (\Sigma_1 \cup \Sigma_2, L_1 \cup L_2) \\ IMP((\Sigma_1, L_1), (\Sigma_2, L_2)) &= (\Sigma_1 \cup \Sigma_2, L_1^c \cup L_2) \end{aligned}$$

where L_1^c *denotes the complement of* L_1 *against* $(\Sigma_1 \cup \Sigma_2)^\omega$.

Example *An example EE is below. The EE specifies a language that allows either of the languages:* EE_{1a} *or* EE_{1b}. *Execution environment* EE_{1a} *specifies that all strings must begin with an* a *and consist of only* a*'s and* b*'s while execution environment* EE_{1b} *specifies that all strings are alternative combinations of a* c *followed by a* b. *Execution environment* EE_1 *specifies a language that allows strings in the set* $a(a+b)^\omega \cup (cb)^\omega$.

$EE_1 = EITHER(EE_{1a}, EE_{1b})$
where
$EE_{1a} = (\{a, b\}, a(a+b)^\omega)$
$EE_{1b} = (\{b, c\}, (cb)^\omega)$

3.4 Intermediate Execution Environments

Execution environments provide a mechanism for encoding a language that acts as a filter on a program. Using configurations, an EE can be transformed from the abstract level to the tool level for multiple verification tools. The ability to link EEs introduces the potential for using intermediate levels to represent an EE between the abstract level and the tool level.

Using intermediate levels allows a policy to be gradually transformed from the abstract level to multiple tool representations. This allows a user's security policy to be maintained

more easily as there is a step by step paper-trail from the abstract level to the tool level specifying the tool specific representation for the user's security requirements. If a user's security requirements change, only the configurations from the abstract level to the middle level need to be changed. Similarly, if a new tool is added, only configurations from the middle level to the tool level need to be created. Furthermore, once the configurations for the new tool have been created, they shouldn't need to be changed. Both of these benefits assume appropriate configurations are devised.

Using intermediate levels also allows for configurations to be reused. Configurations that defined similar mappings from one EE to another can be reused, specifically the configurations mapping an EE into a representation for a specific tool's syntax.

EEs are changed to Intermediate Execution Environments (IEE)s and an IEE is allowed to be mapped using a sequence of configurations. Using a sequence of configurations creates intermediate levels where an IEE is represented prior to being represented in a specific syntax for a particular verification tool. Figure 3.3 displays a pictorial view of this.

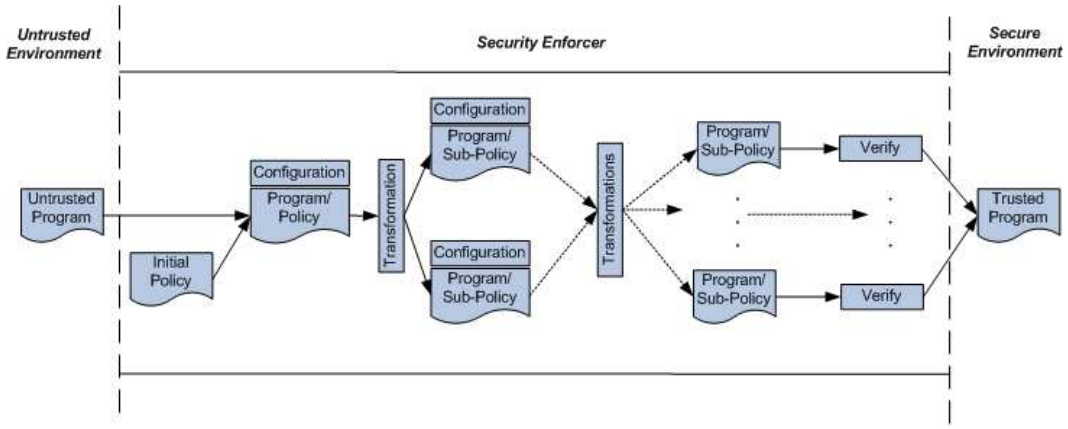

Figure 3.3: Framework using a sequence of configurations.

An IEE has the same structure as an EE, it is simply renamed from an EE to an IEE.

There may not always be a single configuration to apply to a policy. For example, there may be one configuration for mapping a policy for Java programs and another con-

figuration for C programs. The possible configurations to apply to an IEE form a graph of configurations where each node in the graph is a representation of a policy and each edge is a configuration. An example graph is depicted in Figure 3.4.

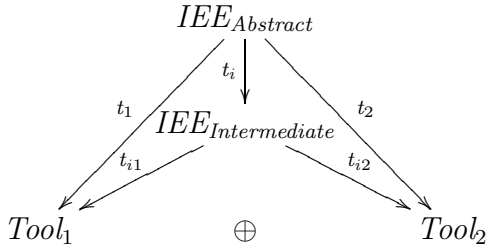

Figure 3.4: Example of a graph that uses configurations to create an intermediate level.

The example graph extends the graph depicted in Figure 3.2 to include an intermediate level. The graph indicates that two configurations can be applied to the IEE. One configuration maps directly to the tool level IEEs using t_1 and t_2. The other maps to an intermediate level first, and then into the two tool level IEEs.

3.4.1 Transforming Intermediate Execution Environments

To transform one IEE to another, a single mapping could be used. To enhance the process of specifying a mapping from one IEE to another, smaller mappings are allowed to be specified and compose mappings together using a number of operators. Using smaller mappings allows configurations to be specified more easily. Furthermore, they allow mappings to be more easily reused. There are many methods for defining a configuration. This section introduces the method used in this dissertation.

A configuration is composed of mappings. The two types of mappings used in this dissertation are described below.

Definition 3.5 (Strings to Strings) *The first type maps strings to strings. This is defined below where* $\Sigma_1, \Sigma_2 \subseteq \Sigma^u$.

$$Maps = \{f \mid f : \Sigma_1^\omega \to \Sigma_2^\omega\}$$

Definition 3.6 (Strings to Languages) *The second type maps strings to languages. This is used to specify a mapping from a single symbol into multiple symbols ranging over a set of values. This is defined below where* $\Sigma_1, \Sigma_2 \subseteq \Sigma^u$.

$$LiftedMaps = \{f \mid f : \Sigma_1^\omega \to \mathbb{P}\,\Sigma_2^\omega\}$$

The definition below shows how the mappings are applied to an IEE.

Definition 3.7 *The function* TIEE *takes a mapping or lifted mapping and an IEE and defines how a mapping and lifted mapping are applied to an IEE. If the mapping maps strings to strings, then each string in the IEE is transformed using the mapping. If the mapping maps strings to languages, each string is transformed using the mapping and the resulting languages are unioned together. The alphabet of the transformed IEE is the alphabet of the range of the mapping function. This is denoted by the function* ranM.

$$TIEE : (Maps \cup LiftedMaps) \to IEE \to IEE$$

$$TIEE\ m\ (\Sigma, L) = \begin{cases} (ranM(m), \{m(w) \mid w \in L\}) & if\ m \in Maps \\ (ranM(m), \bigcup\{m(w) \mid w \in L\}) & if\ m \in LiftedMaps \end{cases}$$

Example *This example demonstrates a mapping from strings to strings. The mapping* f_1 *specifies a mapping from strings with the symbol* b *following the symbol* a *to the symbol* b *with the map of the rest of the string concatenated onto the end, and the symbol* a *not followed by the symbol* b *to the symbol* d *with the map of the rest of the string concatenated onto the end.*

$$f_1 : \{a, b\}^\omega \rightarrow \{b, d\}^\omega$$
$$f_1(a;\ w) = \begin{cases} d;\ f_1(w) & \textit{if}\ \ w(0) \neq b \\ f_1(w) & \textit{otherwise} \end{cases}$$
$$f_1(b;\ w) = b;\ f_1(w)$$

Example *This example demonstrates a mapping from strings to strings where values are used. The function f_2 defines a mapping from the symbol* access(u,v) *for a particular value of* u *and* v *to the symbol* read(u,v) *for the same* u *and* v *with the mapping of the rest of the string concatenated onto the end.*

$$f_2 : \bigcup_{u,v \in Val} \{access(u, v)\}^\omega \rightarrow \bigcup_{u,v \in Val} \{read(u, v)\}^\omega$$
$$\forall\, u, v \in Val \bullet f_2(access(u, v);\ w) = read(u, v);\ f_2(w)$$

Example *This example demonstrates a lifted mapping where a single symbol is mapped into multiple symbols. The function f_{uv} defines a mapping from strings with the symbol* access *to a set of strings where each string maps the symbol* access *into the symbol* read(u,v) *for every value of* u *and* v*.*

$$f_{uv} : \{access\}^\omega \rightarrow \mathbb{P}(\bigcup_{u,v \in Val} \{read(u, v)\})^\omega$$
$$f_{uv}(access;\ w) = \{s_1;\ w_2 \mid w_2 \in f_{uv}(w) \wedge s_1 \in \bigcup_{u,v \in Val} \{read(u, v)\}\}$$

In a configuration, two operators that can be used to combine mappings are: IEE map composition ($\circ_c$) and IEE map shuffle ($||_c$). These operators represent function composition and shuffle respectively but over mappings for IEEs. The semantics for the IEE map composition and shuffle operators use the function composition and shuffle operators respectively. The semantics is defined below. The configuration symbols are denoted by a subscript c.

IEE map composition combines mappings in a configuration together specifying that both mappings are applied to an IEE one after the other. The IEE map composition operator is right associative. The IEE map composition operator is defined in terms of the function composition operator ($\circ$) defined in Definition 3.8.

Definition 3.8 (Function Composition)

$$\circ : (Maps \cup LiftedMaps) \rightarrow (Maps \cup LiftedMaps) \rightarrow IEE \rightarrow IEE$$
$$t_1 \circ t_2\ e = (TIEE\ t_1)\ ((TIEE\ t_2)\ e)$$

The IEE map shuffle operator combines two mappings in a configuration where each mapping is applied to an IEE separately and the resulting languages are *shuffled* together. The IEE map shuffle operator is defined in terms of the shuffle operator ($||_t$) defined in Definition 3.9. The shuffle of two sentences is the set containing all arbitrary interleavings of the two sentences. An example shuffle of two regular expressions is

$$ab \mid\mid cd = \{abcd, acbd, acdb, cabd, cadb, cdab\}$$

The shuffle of two languages (denoted by the $||_l$ symbol) is the set of all strings obtained by shuffling all strings of one language with all strings from the other.

Definition 3.9 (Shuffle)

$$||_t : (Maps \cup LiftedMaps) \rightarrow (Maps \cup LiftedMaps) \rightarrow IEE \rightarrow IEE$$
$$t_1\ ||_t\ t_2\ e = ((TIEE\ t_1)\ e)\ ||_l\ ((TIEE\ t_2)\ e)$$
where
$$(\Sigma_1, L_1)\ ||_l\ (\Sigma_2, L_2) = (\Sigma_1 \cup \Sigma_2, \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1\ \mid\mid\ w_2)$$
$$s_1;\ w_1\ \mid\mid\ s_2;\ w_2 = \{s_1;\ w_a \mid w_a \in (w_1 \mid\mid s_2;\ w_2)\} \cup \{s_2;\ w_b \mid w_b \in (s_1;\ w_1 \mid\mid w_2)\}$$

To allow the specifier simplicity when splitting representations, a split is explicitly allowed to be specified in a configuration. A split is specified using *link operators*. Each

link operator corresponds to an IEE operator. Thus, for the operators used in an IEE in this dissertation, three link operators are included in a configuration: $BOTH_c$, $EITHER_c$ and IMP_c. Each link operator in a configuration takes two sub-configurations. The sub-configurations specify how an IEE is going to be transformed and the two results linked together using the specified operator.

The syntax for a configuration used in this dissertation is given below. The semantics of this configuration are given in Definition 3.10 below. If additional IEE operators are included then additional link operators will need to be included in a configuration and given appropriate semantics.

$$
\begin{aligned}
\mathit{CompMappings} = \quad & m \in \mathit{Maps} \\
& \mid lm \in \mathit{LiftedMaps} \\
& \mid \mathit{CompMappings} \circ_c \mathit{CompMappings} \\
& \mid \mathit{CompMappings} \ ||_c \ \mathit{CompMappings} \\
\mathit{Config} = \quad & \mathit{CompMappings} \\
& \mid \mathit{BOTH}_c(\mathit{Config}, \mathit{Config}) \\
& \mid \mathit{EITHER}_c(\mathit{Config}, \mathit{Config}) \\
& \mid \mathit{IMP}_c(\mathit{Config}, \mathit{Config})
\end{aligned}
$$

Definition 3.10 *The semantics of a configuration are below where* $m \in \mathit{Maps}$, $lm \in \mathit{LiftedMaps}$, $cm_1, cm_2 \in \mathit{CompMappings}$ *and* $c_1, c_2 \in \mathit{Config}$.

$$
\begin{aligned}
[\![m]\!] &= TIEE(m) \\
[\![lm]\!] &= TIEE(lm) \\
[\![cm_1 \circ_c cm_2]\!] &= [\![cm_1]\!] \circ [\![cm_2]\!] \\
[\![cm_1 \ ||_c \ cm_2]\!] &= [\![cm_1]\!] \ ||_t \ [\![cm_2]\!] \\
[\![BOTH_c(c_1, c_2)]\!] &= BOTH([\![c_1]\!], [\![c_2]\!]) \\
[\![EITHER_c(c_1, c_2)]\!] &= EITHER([\![c_1]\!], [\![c_2]\!]) \\
[\![IMP_c(c_1, c_2)]\!] &= IMP([\![c_1]\!], [\![c_2]\!])
\end{aligned}
$$

Example *This example specifies a simple configuration containing a single mapping and shows how an IEE is mapped using this configuration. The configuration $t_1 = f_1$ specifies that the function f_1 is applied to a given IEE. Consider the IEE*

$IEE_2 = EITHER(IEE_{2a}, IEE_{2b})$
where
$IEE_{2a} = (\{a, b\}, (a)^\omega)$
$IEE_{2b} = (\{a, b\}, (ab)^\omega)$

which specifies that all strings must consist of only a*'s or must consist of alternating combinations of an* a *followed by a* b*. Performing the transformation t_1 EE results in the IEE*

$IEE_2 = EITHER(IEE_{2a}, IEE_{2b})$
where
$IEE_{2a} = (\{b, d\}, d^\omega)$
$IEE_{2b} = (\{b, d\}, b^\omega)$

Example *This example demonstrates a transformation of a configuration that contains a lifted mapping. The example configuration $t_{1b} = f_{uv}$ specifies that an IEE is to be transformed using the mapping f_{uv}. The configuration maps strings of the symbol* access *to strings of the symbols* read(u,v) *for all values of* u *and* v*. Performing the transformation*

t_{1b} $(\{access\}, access^{\omega})$ *would result in the IEE*

$$(\bigcup_{u,v \in Val} \{read(u,v)\}, (\bigoplus_{u,v \in Val} \{read(u,v)\})^{\omega})$$

Example *This example demonstrates the use of function composition. The configuration* t_2 *below specifies that* f_1 *is applied to a given IEE first and then* f_3 *is applied to the result of the first transformation. The function* f_3 *maps the symbol* b *to the symbol* c *and leaves the symbol* d *as is.*

$t_2 = f_3 \circ_c f_1$
where
$f_3 : \{b, d\}^{\omega} \rightarrow \{c, d\}^{\omega}$
$f_3(b;\ w) = c;\ f_3(w)$
$f_3(d;\ w) = d;\ f_3(w)$

Performing the transformation t_2 $(\{a, b\}, (a+ab)^{\omega})$ *results in the IEE* $(\{c, d\}, (d+c)^{\omega})$.

Example *This example demonstrates the use of the shuffle operator to map a string consisting of only* a*'s to a set of strings containing all the possible interleavings of the symbols* a *and* d. *The configuration,* $t_3 = f_1 \,||_c\, f_4$ *where* f_4 *is defined below, specifies that an IEE is transformed using* f_1 *and* f_4 *independently and the results are shuffled together.*

$f_4 : \{a, b\}^{\omega} \rightarrow \{a, c\}^{\omega}$
$f_4(b;\ w) = c;\ f_4(w)$
$f_4(a;\ w) = a;\ f_4(w)$

Applying the transformation t_3 $(\{a, b\}, a^{\omega})$ *results in the IEE* $(\{a, d\}, (a + d)^{\omega})$.

Example *This example demonstrates a configuration that uses a link operator to split an IEE. The configuration* $t_4 = EITHER_c(f_1, f_4)$ *specifies that a given IEE is transformed using* f_1 *and* f_4 *independently and then combined together using the EITHER operator. Performing the transformation* t_4 $(\{a, b\}, (a + ab)^{\omega})$ *results in*

$EITHER((\{d, b\}, (d + b)^\omega), (\{a, c\}, (a + ac)^\omega))$

3.5 Implementation

A Haskell implementation of the framework presented in this chapter will be discussed in this section. The implementation represents languages as LTL formula. While the language accepted by an LTL formula is a set of strings, the implemented translator is only a subset of the framework as Büchi automata are more expressive than LTL formula. The version however is sufficient enough to demonstrate the framework.

3.5.1 IEEs

A temporal logic formula is implemented as the data type *TLFormula* which contains the usual temporal logic operators (*always*, *eventually*, *next*) over the data type *predicate*. A predicate is implemented using the usual Boolean operators (*not*, *and*, *or* and *implies*) over atomic predicates.

An IEE is implemented as the data type

data type IEE = *Language TLFormula*
| *IAND IEE IEE*
| *IOR IEE IEE*
| *IIMP IEE IEE*

An IEE contains four subtypes. The subtype *Language* represents a single language which is a temporal logic formula. The subtype *IAND* represents a combination of two IEEs joined together using the *BOTH* operator. The subtype *IOR* represents a combination of two IEEs joined together using the *EITHER* operator and the subtype *IIMP* represents a combination of two IEEs joined together using the *IMP* operator. Additional subtypes can be added to this data types for any additional operators a user wants to include in an IEE.

3.5.2 Configurations

Composed mappings are implemented as *rule*s. There are two types of rules: *TLMap* and *PMap* shown below.

$$type\ TLMap = (TLFormula, TLFormula)$$
$$type\ PMap = (Variable, [Value])$$

TLMap is a type that defines a mapping from one formula to another. The *TLMap* rule implements a mapping from languages to languages. The first element specifies the domain of a function and the second specifies the range of the function. In a *TLMap* rule variables can be used in the atomic predicates. Variables are used to specify a particular set of values from a set.

A *PMap* rule allows mappings from a variable to a set of values to be specified. This imitates the functionality of the lifted mapping functions.

A *Rule* is a list of these two types. A list is used to represent the composition of functions. A configuration is implemented as a data type of four subtypes shown below.

$$\begin{aligned} data\ type\ CONF &= Rules\ Rule \\ &\mid CAND\ CONF\ CONF \\ &\mid COR\ CONF\ CONF \\ &\mid CIMP\ CONF\ CONF \end{aligned}$$

3.5.3 The Transformer

The transformer accepts a Haskell list of configurations and an IEE and transforms the IEE using the given configurations. The transformer consists of three main mapping functions for transforming an IEE: *transIEE*, *transTLFormula* and *transVar*.

The *transIEE* function takes a single configuration and an IEE and transforms the IEE. If the configuration is of the types *CAND*, *COR* or *CIMP*, the function transforms the given IEE using the two sub-configurations independently and then joins the results

using the appropriate operator, for example, *IAND* for *CAND*. If the configuration is of type *Rules*, the *transIEE* function calls either the *transTLFormula* or *transVar* function for each of the rules in the list of rules, for each language in the IEE.

The *transTLFormula* function transforms a language using a *TLMap* rule. All occurrences of the formula in the domain of a *TLMap* rule in a given language are mapped into its range.

The *transVar* function transforms a language using a *PMap* rule. That is, that function transforms a given language such that any occurrence of the variable in the domain of the rule that appears in the given language is mapped into the disjunction of the values in the range of the rule.

Chapter 4

Case Studies and Examples

The framework described in the previous chapter has been tested on a number of case studies. Each case study demonstrates how an initial abstract policy is represented as an IEE and then mapped using a sequence of configurations to generate a number of tool specific representations for that policy.

The first case study shows how multiple tool level representations can be generated from an initial policy. Depending on the given program, a different sequence of configurations can be applied to the IEE. The second case study involves threads and shows how the abstract policy of thread atomicity is transformed into a tool specific implementation. The final case study is much more complex and involves a mobile phone program which simulates the TV game show "Who wants to be a millionaire?". The case study shows how an initial IEE is transformed into a specific tool for verification.

The case studies now deal with programs. Since it's easier to transform programs at the syntactic level, the main configurations are described at the syntactic level. The definition below shows how syntactic mappings are consistent with the framework.

Definition 4.1

$f : (\Sigma_1, L_1) \rightarrow (\Sigma_2, L_2)$
where
$L_1 = \mathcal{L}(Pgm_1)$
$L_2 = \mathcal{L}(Pgm_2)$

The transformer transforms Pgm_1 to Pgm_2 where Pgm_2 is the annotated program. Therefore, the language of an IEE is the language induced by an annotated program, denoted $\mathcal{L}(Pgm)$ where Pgm is a given annotated program.

4.1 Memory Case Study

The Memory case study demonstrates how multiple tool representations can be generated from an initial representation. The initial user's security requirements are "*To be safe, a program must be memory safe*". Memory safety is a very abstract term that differs depending on the implementation of a given program. In this example, two implementations of memory are considered: arrays and stacks. Depending on which implementation(s) of memory a program implements, a different sequence of configurations is applied to the initial IEE to generate an IEE specific for that program.

At the high level, an IEE for representing that only safe memory is allowed to be accessed is

$$IEE_{mem} = (\{memorySafe, notMemorySafe\}, memorySafe^{\omega})$$

The IEE consists of two symbols: *memorySafe* which represents all memory safe operations and *notMemorySafe* which represents all unsafe memory operations. The IEE specifies that all strings must contain only *memorySafe* symbols.

The symbol *memorySafe* is very abstract and can have many tool level representations.

At one level, memory safety can be viewed as safe read and write operations. Alternatively at another level, memory safety can be viewed as pushing an element onto a stack that is not full and popping an element from a stack that is not empty. The next section defines a number of configurations to map IEE_{mem} through a number of intermediate levels into a number of tool specific representations. By using intermediate levels, additional tool level representations can be easily added.

4.1.1 Configurations

First, a configuration is specified to map the abstract terms *memorySafe* and *notMemorySafe* respectively into safe and unsafe, read and write operations. This creates an intermediate level between the abstract representation for accessing memory and the tool specific representations. The configuration t_{rw} below defines a mapping from the initial IEE into the intermediate level just described.

In this chapter, the mappings for each symbol are specified in tables for improved readability. The left hand column of the table specifies the arguments to the function while the right hand column specifies the result.

$t_{rw} = f_r \;||_c\; f_w$

f_r :

memorySafe	*safeRead*
notMemorySafe	*unsafeRead*

f_w :

memorySafe	*safeWrite*
notMemorySafe	*unsafeWrite*

The configuration consists of two mappings shuffled together. The function f_r maps the symbols *memorySafe* to *safeRead* and *notMemorySafe* to *unsafeRead* in a string, while the function f_w maps the symbols *memorySafe* to *safeWrite* and *notMemorySafe* to *unsafeWrite* in a string. By shuffling these mappings together, a string consisting

of *memorySafe* symbols is mapped into multiple strings containing the *safeRead* and *safeWrite* symbols in any combination.

The next configuration t_{axi} refines both the safe and unsafe, read and write operations to take three arguments: a memory storage (a), a location in that memory storage for data to be read from or written to (i), and either the value to be written or the variable in which to store the read data (x).

In the table below, the results of each mapping are specified as a set of symbols. Using a set of symbols in the right hand column, a mapping from a single symbol to a set of symbols is specified.

t_{axi} :

$safeRead$	$\bigcup_{a,x,i \in Val} \{safeRead(a,x,i)\}$
$unsafeRead$	$\bigcup_{a,x,i \in Val} \{unsafeRead(a,x,i)\}$
$safeWrite$	$\bigcup_{a,x,i \in Val} \{safeWrite(a,x,i)\}$
$unsafeWrite$	$\bigcup_{a,x,i \in Val} \{unsafeWrite(a,x,i)\}$

The next configuration t_{srw} below is used to map an IEE such that the safe read and write operations are mapped to have a *check* prior to the operations to ensure the respective operation is safe. At this stage, what exactly is being checked remains abstract as it depends on the specific implementations of reading and writing to memory. Therefore, checks are made on the memory storage (a) and the index value (i) used to access memory.

Quantification is used in the table to denote the mapping of all the symbols ranging over a set values. This simplifies the process of individually specifying a mapping for each symbol, for all values.

t_{srw} :

$\forall\, a, x, i \in Val \bullet safeRead(a, x, i)$	$checkRead(a, i).read(a, x, i)$
$\forall\, a, x, i \in Val \bullet unsafeRead(a, x, i)$	$read(a, x, i)$
$\forall\, a, x, i \in Val \bullet safeWrite(a, x, i)$	$checkWrite(a, i).write(a, x, i)$
$\forall\, a, x, i \in Val \bullet safeRead(a, x, i)$	$write(a, x, i)$

Depending on the implementation of a program, reading and writing to memory can have a number of implementations.

Safe Array Access

For memory represented as an array, read operations are the assignment of an element of an array at some index to a variable while write operations are the assignment of some value to an array at some index. A safe read or write operation for an array is accessing an array within the bounds of the array. It is assumed that array indices begin at 0. The Java notation *a.length* is adopted to denote the length of an array *a*. Accessing an array within the bounds of an array is accessing an index point that is greater than or equal to zero and less than the size of the array. The configuration t_{array} below defines a mapping from the read and write operations for an array implementation.

t_{array} :

$\forall\, a, x, i \in Val \bullet read(a, x, i)$	$(x = a[i])$
$\forall\, a, i \in Val \bullet checkRead(a, i)$	$check(0 \leq i \;\&\&\; i < a.length)$
$\forall\, a, x, i \in Val \bullet write(a, x, i)$	$(a[i] = x)$
$\forall\, a, i \in Val \bullet checkWrite(a, i)$	$check(0 \leq i \;\&\&\; i < a.length)$

Safe Stack Access

For stacks, read operations are viewing an element from the top of a stack while write operations are pushing or popping an element to or from a stack. A safe push is pushing an element onto a stack that is not full. Depending on the implementation, e.g., a stack

implemented as a linked list, there may not be a maximum size. A safe pop is popping an element from a stack that is not empty. The configuration t_{stack} below defines a mapping from the read and write operations for a stack implementation.

t_{stack} :

$\forall\, a, i \in Val \bullet read(a, x, i)$	ε
$\forall\, a, i \in Val \bullet checkRead(a, i)$	ε
$\forall\, a, x, i \in Val \bullet$ $checkWrite(a, i).write(a, x, i)$	$(check(!isFull(a)).push(a, x))$ $+(check(!isEmpty(a)).pop(a, x))$

Safe Array and Stack Access

Some programs may implement both an array and a stack. Therefore, safe memory access is safe array access and safe stack access. The configuration t_{as} below specifies a mapping from the read and write operations for a stack and an array implementation.

$$t_{as} = BOTH_c(t_{stack}, t_{array})$$

Configurations for Verification Tools

The configurations for mapping an IEE into specific syntaxes for a number of verification tools are now defined. For this case study, the tool Blast is used to verify C programs and the tools Bandera and Java PathFinder are used to verify Java programs.

For the tools Blast and Java PathFinder, checks can be encoded as *assert* statements. The configuration t_{assert} below defines a mapping from the symbols $check(c)$ for some value c into the symbol $assert(c)$. By using values, the configuration can be reused for all mappings from checks to asserts.

t_{assert} :

$\forall\, c \in C \bullet check(c)$	$assert(c)$

In the configuration t_{assert}, the set C contains all the values for the checks. For this case study, the set C is defined below for an array and stack respectively.

$$C_{array} = \bigcup_{a,i \in Val} \{0 \leq i \ \&\& \ i < a.length\}$$

$$C_{stack} = \bigcup_{a \in Val} \{!isEmpty(a)\} \cup \bigcup_{a \in Val} \{!isFull(a)\}$$

For the Bandera tool, checks are represented using assert statements but in a different syntax to Blast and Java PathFinder. In Bandera, assertions have the form

$$/ * **@assert * POSTInv :< assert >; */$$

where $< assert >$ is a particular boolean expression that is going to be checked by Bandera. The configuration $t_{bandera}$ below defines a mapping from a *check* into a Bandera syntax assert statement. Similar to the configuration t_{assert}, the configuration $t_{bandera}$ uses values to make the configuration reusable.

$$t_{bandera} : \begin{array}{|l|l|} \hline \forall\, c \in Val \bullet check(c) & / * **@assert * POSTInv : c; */ \\ \hline \end{array}$$

4.1.2 Memory Verification

Depending on the program being verified, different sequences of configurations can be applied to IEE_{mem}. Figure 4.1 shows a graph of the possible sequences of configurations.

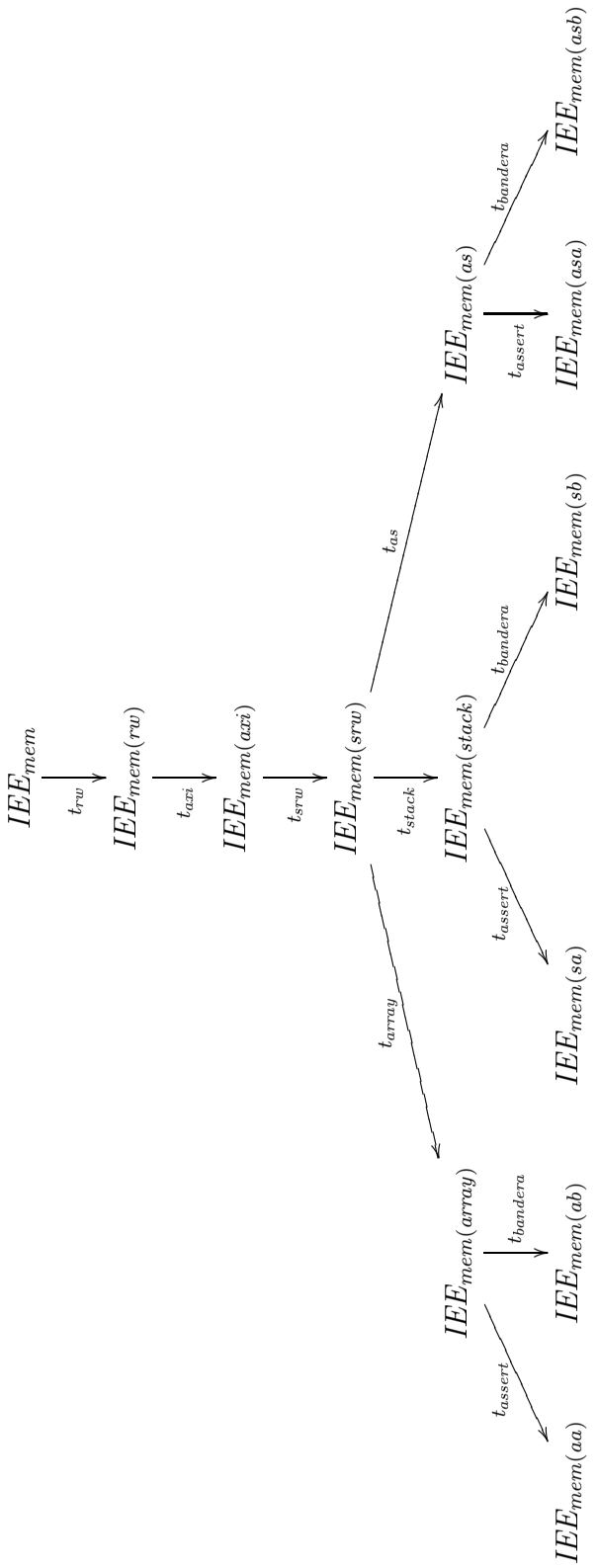

Figure 4.1: Graph of possible configurations for safe memory. The

At the tool level in Figure 4.1, the letters in the brackets represent the type of memory implementation the IEE is specifying (a for array, s for stack and as for both an array and stack) and the tool(s) that can be used to verify a program using the IEE (a for assert, so either JPF or Blast depending on whether the program is in C or Java respectively, and b for Bandera). For example, the intermediate execution environment $IEE_{mem(ab)}$ will be used by Bandera to verify programs that implement memory as an array.

For this case study, four C and four Java programs were implemented. The Java programs implemented have the same functionality as the C programs. Two programs were implementations to test array verification: one for safe array access and the other for unsafe array access. Similarly, two programs were implementations to test stack verification: one for safe stack access and the other for unsafe stack access.

For the two C programs that implemented memory as a stack, the sequence $t_{assert} \circ_c t_{stack} \circ_c t_{srw} \circ_c t_{axi} \circ_c t_{rw}\ IEE_{mem}$ was used. For the two Java programs that implemented memory as a stack, the sequence of configurations $t_{bandera} \circ_c t_{stack} \circ_c t_{srw} \circ_c t_{axi} \circ_c t_{rw}\ IEE_{mem}$ was used. For the two C programs that implemented memory as an array the sequence of configurations $t_{assert} \circ_c t_{array} \circ_c t_{srw} \circ_c t_{axi} \circ_c t_{rw}\ IEE_{mem}$ was used and for the two Java programs that implemented memory as an array the sequence of configurations $t_{bandera} \circ_c t_{array} \circ_c t_{srw} \circ_c t_{axi} \circ_c t_{rw}\ IEE_{mem}$ was used.

For each program, the syntactic function was used to automatically insert the assertions transformed from the configurations into a program. Applying this function, the alphabet of the IEE becomes the alphabet of the transformed IEE plus the alphabet of the program, and the language of the IEE becomes the language induced by the annotated program resulting from the syntactic function.

Given an annotated program, all tools were successful in verifying the safe implementations and not verifying the unsafe implementations.

4.2 Bank Case Study

The second case study is a Java program involving threads that simultaneously access a user's Bank account. With multiple users accessing the account simultaneously, there is a possibility that resources will be interfered with.

The Java program consists of an account and methods for crediting and debiting money to and from the account. The pseudo code for crediting and debiting money is shown in Figure 4.2.

```
debit(amount)
    if(balance − amount >= 0)
    then
        balance = balance − amount
    else
        error
```

```
credit(amount)
    balance = balance + amount
```

Figure 4.2: Pseudo code for crediting and debiting money to and from an account.

The importance of thread atomicity is shown in the scenario presented in Figure 4.3 where two debtors attempt to debit an amount of money from the same account at the same time. In the example, assume the starting balance is *$110*.

$debtor1 : debit(\$100)$
$debtor1 : if(\$110 > \$100)$
$debtor2 : debit(\$20)$
$debtor2 : if(\$110 > \$20)$
$debtor1 : balance = \$110 - \$100$
$debtor2 : balance = \$10 - \$20$

Figure 4.3: Scenario that involves thread interference.

This scenario is neither the desired nor the expected computation. The desired and expected outcome is that one transaction will succeed and the other will fail due to insufficient funds. Instead, both debtors were able to debit money from the account resulting in

a negative balance. With thread interference it is possible for money to disappear or even appear.

Ensuring the atomicity of actions is important. The IEE below specifies a policy that allows only atomic actions to be performed.

$$IEE_{bank} = (\{atomicAction, nonAtomicAction\}, atomicAction^{\omega})$$

There are many ways to ensure thread atomicity. For this case study the intermediate execution environment IEE_{bank} is mapped through a number of intermediate levels into a tool specific representation. Through each intermediate level, both the actions that are atomic and the representations of atomic actions at the tool level are refined.

4.2.1 Configurations

A number of configurations are defined to map the intermediate execution environment IEE_{bank} into tool specific representations.

First, a configuration specifying the actions that need to be atomic are defined. In this case, the actions *debit* and *credit* are to be atomic. The configuration t_{cd} below defines this mapping.

$t_{cd} = f_c \,||_c\, f_d$

f_c :	*atomicAction*	*atomicCredit*
	nonAtomicAction	*nonAtomicCredit*
f_d :	*atomicAction*	*atomicDebit*
	nonAtomicAction	*nonAtomicDebit*

This configuration consists of two mappings joined together using the shuffle operator. The configuration maps strings consisting of *atomicAction* symbols into strings containing

any combination of *atomicCredit* and *atomicDebit* symbols.

Next, what it means for the *debit* and *credit* actions to be atomic are defined. For this, the configuration t_{aa} is used to map the symbols *atomicCredit* and *atomicDebit* into the sequences $p.credit.v$ and $p.debit.v$ respectively. The symbols p and v are abstract symbols used to represent acquiring a resource and releasing a resource respectively.

t_{aa} :

atomicCredit	$p.credit.v$
nonAtomicCredit	*credit*
atomicDebit	$p.debit.v$
nonAtomicDebit	*debit*

A number of mechanisms can be used to ensure the atomicity of an action. For Java, actions are atomic if the operation is encapsulated within a *synchronized* block. The configuration t_{syn} defines a mapping from p to the beginning of a synchronization block and v to the close of the block.

t_{syn} :

p	*synchronized*{
v	}

4.2.2 Verification of Thread Atomicity

For this case study, the intermediate execution environment IEE_{bank} was mapped using the sequence

$$t_{syn} \circ_c t_{cd} \circ_c t_{aa}\ IEE_{bank}$$

This mapped IEE specifies that for Java programs, all *debit* and *credit* method calls need to be encapsulated within a synchronized block. Using Java itself as a syntactic checking tool, the Java *bank* program was verified to ensure all credit and debit method

calls were encapsulated within synchronized blocks.

4.3 Millionaire Case Study

The final case study aims at lowering the restrictions placed on a mobile phone for a particular application. The application of interest is *Millionaire*, a computer application of the game "Who Wants to be a Millionaire" for mobile phones. The game supports certain features that would usually be seen as a security risk. This case study shows how the security concerns apparent in the application can be expressed in the framework presented in this dissertation, regardless of the implementation of the program. Given a specific implementation of the Millionaire application, the method for transforming the security requirements into usable representations is described below.

By initially representing the security requirements independent of the implementation of the application, that representation can be reused for all implementations of the application. This is possible because the security concerns are the same for all implementations of the application. Thus, the initial encoding of the security requirements does not need to be changed for new implementations of the program. However, the implementation of certain program features can vary according to the specific implementation the program. Therefore, the exact process of verification depends on the implementation of the program, but *only* the configurations mapping the high level security requirements into tool specific representations need to be changed for each new implementation.

For this case study, the Millionaire computer game is implemented in Java. Given this Java implementation, the IEE specifying the security concerns rising from the Millionaire application is transformed so specific verification tools can be used to determine whether the Java program satisfies these security concerns. The verification tools used in this case study are Java PathFinder (JPF) and Bandera, which were chosen because of the author's

familiarity with them.

4.3.1 *Millionaire* Background

The Millionaire application has been adapted from the TV game show "Who Wants to be a Millionaire." The game consists of answering a number of consecutive questions. Each question has four multiple choice answers, one of which is correct. When a player answers a question correctly, they move on to the next question. If the player answers a question incorrectly, the game ends. The goal of the game is to answer as many questions as possible. The more questions answered correctly, the higher the prize money incentive.

To help the player with harder questions, the game offers lifelines. Each player has three lifelines: *50:50*, *ask the audience* and *phone a friend*. Each lifeline can only be used once. The *50:50* lifeline removes all but two of the multiple choice answers, one correct and one incorrect. Thus the player has a fifty percent chance of answering the question correctly.

The *ask the audience* lifeline is a poll taken by the audience. The poll asks the audience which multiple choice option they think is correct. The results of the poll are then given to the player. This allows the player to know what the audience thinks the answer is.

Finally, the *phone a friend* lifeline allows the player to call one friend. The player has thirty seconds to communicate with the friend. Usually this communication involves the player telling the friend the question and the possible answers, followed by the friend providing their opinion regarding the correct answer.

4.3.2 Adaptation of *Millionaire*

In this case study, the TV game show "Who Wants to be a Millionaire" is implemented as a computer application for mobile phones.

The *Millionaire* game for mobile phones used in this case study uses the same principles as "Who Wants to be a Millionaire." The main difference between the TV game show and the mobile phone application is that the player plays on a mobile phone instead of in a television studio. As a result of this difference, a few changes are made to the game. The main difference is that only two lifelines are given in the mobile phone application. The reason for this is that the *ask the audience* lifeline is not included because there is no audience involved when playing on a mobile phone.

The application consists of a start up phase that collects information from the player and initialises the game. During the start up phase of the application, the player is asked to enter two friends for the *phone a friend* lifeline. If the friend is not already in the player's address book, the name and number of that friend is requested and then entered into the address book. When the *phone a friend* lifeline is used, the player is asked to selected one of the two friends given in the start up phase. The *Millionaire* application then looks up their number in the player's address book and calls the friend.

One of the main issues identified in the *Millionaire* mobile phone game is the potential for a player to play outside the parameters of the game. This can occur by a player obtaining information from someone without the use of a lifeline. To prevent this, players are not allowed to phone or message anyone while playing the game. During the start up phase of the *Millionaire* application, the phone's ability to make phone calls or send messages is disabled. By disabling these features, players are unable to use them, which helps to ensure that players play within the parameters of the game. Once the application has concluded, the features are re-enabled.

As it is impractical to ensure that no one is physically near the player and helping them during the game, it is assumed that players play alone. Furthermore, it is assumed that the player does not have other devices, such as a computer, second mobile phone or PDA with them.

4.3.3 Security Concerns

The *Millionaire* application uses certain features which access secure mobile phone features. These include managing an address book, making a telephone call, and enabling and disabling mobile phone features. A number of security concerns arise from these features. These security concerns need to be addressed so recipients of the *Millionaire* application can know whether the application is safe or whether it will harm their mobile phones.

Most of the security concerns arise from the *phone a friend* lifeline. The *phone a friend* lifeline performs three tasks: entering new information into the address book (optional), obtaining telephone numbers from the address book and making a telephone call. All of these tasks involve either accessing phone memory or the hardware of the phone.

With regards to entering new information into the address book, the main security concerns are that no information in the address book is overwritten (P1) and that at most two names are added into the address book (P2).

With regards to obtaining a telephone number, the security concern is that only a friend's number, selected during the start up phase of the application, is looked up and that no other information is accessed (P3).

With regards to making a telephone call, the security concerns are that at most one phone call is made (P4) and only a friend's number is dialed (P5).

The other security concerns arise from the application's mechanism for ensuring players play within the parameters of the game. More precisely, security concerns arise from enabling and disabling the message and telephone features. The first security concern which arises is ensuring that only the message and telephone features are disabled (P6). The second security concern arises from ensuring that if either of these features are disabled, then at some point before the application ends, they are enabled (P7).

Table 4.1 shows a summary of the security relevant features for the *Millionaire* application.

Name	Description
P1	No overwriting of information in the address book
P2	No more than two names are added into the address book
P3	Only friends information is looked up in the address book
P4	Only one phone call is made
P5	Only a friend's number is dialed
P6	Only the message and telephone features are disabled
P7	Disabled features are restored to their original settings

Table 4.1: Summary of security relevant issues

4.3.4 *Millionaire* Security Policy

There are a number of ways the security concerns mentioned in the previous section can be represented in the framework presented in this dissertation. One method is to encode all the security concerns as a single IEE. An alternative method is to encode each security concern as a separate IEE and use the *BOTH* linking operator described in section 3.3.1 to link all the security concerns. This method is depicted in Figure 4.4.

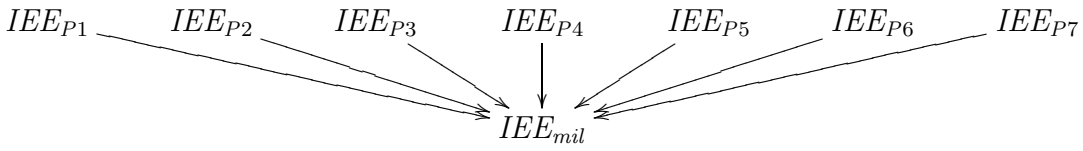

Figure 4.4: Graph showing each security concern represented as a separate IEE

The second method is much more appealing because of the flexibility if any of the security concerns change. A change in the security concerns can be isolated so that IEEs representing the other security concerns are not affected. Furthermore, the IEEs can be freely combined in a way that appropriately represents the user's security requirements. An IEE for each of the security concerns, as listed in Table 4.1 above, is described in the following paragraphs.

Security concern P1 specifies that information in the address book cannot be overwritten. There are four ways an address book can be accessed: add, delete, search and change. These four operations form the basis of the alphabet for the IEE for security concern P1.

This alphabet is below.

$$\Sigma_1 = \{add, change, delete, search\}$$

By preventing any changes or deletions of information in the address book, security concern P1 is captured. IEE IEE_{P1} specifies a language that allows only adding and searching of an address book.

$$IEE_{P1} = (\Sigma_1, (add + search)^\omega)$$

Security concern P2 specifies that at most two names are added into the address book. While a language could be specified allowing at most two add operations, this is very restrictive. If the security requirements change then the language needs to be altered. Instead, security concern P2 can be captured by placing a check before any add symbol. The check, represented by the symbol *checkAdd*, is an abstract symbol which allows the restrictions on the add operation to be changed without having to change the language for the IEE. IEE IEE_{P2} below specifies a language that allows sequences of checking an add followed by an add operation.

$$\Sigma_2 = \{add, checkAdd\}$$
$$IEE_{P2} = (\Sigma_2, (checkAdd\ \ add)^\omega)$$

Security concern P3 specifies that only friends' information is looked up in the address book. Similar to security concern P2, security concern P3 is captured by placing a check before any search operation is made. The symbol *checkSearch* is an abstract symbol used to represent the restrictions on the search operation. IEE IEE_{P3} below captures security concern P3 by specifying a language where a check is performed before every search operation.

$\Sigma_3 = \{search, checkSearch\}$
$IEE_{P3} = (\Sigma_3, (checkSearch\ search)^\omega)$

Security concern P4 specifies that the application makes at most one phone call while security concern P5 specifies that only a specified friend is dialed. For these security concerns a *call* symbol is introduced indicating the program making a phone call. To capture security concerns P4 and P5, a single check can be used before any call operation. The symbol *checkCall* is used to represent this check. Similar to the *checkAdd* and *checkSearch* symbols, the *checkCall* symbol is left abstract. Security concerns P4 and P5 are captured in a single IEE, IEE_{P45} below, that specifies a language that allows sequences of the symbols $checkCall\ call$.

$\Sigma_{45} = \{call, checkCall\}$
$IEE_{P45} = (\Sigma_{45}, (checkCall\ call)^\omega)$

Security concern P6 ensures that only the message and telephone features are disabled on a player's phone. For this security concern symbols are introduced that represent disabling features. The symbols $\bigcup_{f \in Val} \{disableFeature(f)\}$ are used to denote the disabling of all features f.

Security concern P6 states that only the message and telephone features are allowed to be disabled at the start of the application. The symbol *init* is used to denote the start of the application. To specify security concern P6, the message and telephone features are allowed to be disabled in any order. The symbols $disableFeature(msg)$ and $disableFeature(tel)$ are used to denote the disabling of the message and telephone features respectively.

IEE IEE_{P6} represents security concern P6. The alphabet for this IEE includes the symbols $\bigcup_{f \in Val} \{disableFeature(f)\}$ for disabling features and the symbol *init*.

$$\Sigma_6 = \{init\} \cup \bigcup_{f \in Val} \{disableFeature(f)\}$$

The language the IEE specifies allows the message and telephone features to be disabled after the symbol *init*.

$$IEE_{P6} = (\Sigma_6, (init\ ((disableFeature(msg)\ disableFeature(tel))+ \\ (disableFeature(tel)\ disableFeature(msg)))^{\omega})$$

The final security concern, P7, specifies that the message and telephone features are enabled at some point before the application ends. The symbols $\bigcup_{f \in Val} \{enableFeature(f)\}$ represent the enabling of each feature f. Before the end of the application, denoted by the symbol *exit*, the message and telephone features must be enabled. The alphabet Σ_6 below includes the symbols $\bigcup_{f \in Val} \{enableFeature(f)\}$ for enabling features and the symbol *exit*.

$$\Sigma_7 = \{exit\} \cup \bigcup_{f \in Val} \{enableFeature(f)\}$$

The IEE for security concern P7 is below.

$$IEE_{P7} = (\Sigma_7, ((enableFeature(msg)\ enableFeature(tel))+ \\ (enableFeature(tel)\ enablefeature(msg))) \\ exit)^{\omega})$$

Given that the telephone feature was disabled at the beginning of the application, to make a phone call this feature must be temporarily enabled. Furthermore, after a phone call, this feature must be disabled once again. IEE IEE_{P45} is refined to include this.

$$\Sigma_{45} = \{call, checkCall\} \cup \bigcup_{f \in Val} \{enableFeature(f)\} \cup \bigcup_{f \in Val} \{disableFeature(f)\}$$
$$IEE_{P45} = (\Sigma_{45}, (enableFeature(tel)\ checkCall\ call\ disableFeature(tel))^{\omega})$$

Currently, the language of each of the six IEEs only includes the symbols specific to that IEE. For example, IEE IEE_{P2} specifies a language that allows infinite sequences of the symbols *checkAdd add*. Given a program that performs numerous operations including searching and calling, for example, the program will not adhere to the IEE IEE_{P2}. This is because the other symbols allowed in the Millionaire application are not being specified in the language of the IEE.

The language of each of the IEEs above is extended to include the symbols present in the Millionaire application such that the IEEs still capture the security concerns for that IEE. A prime is used to denote the extension of the non-primed IEEs. The primed IEEs are shown below. The alphabet for all IEEs is the union of all the alphabets. This is specified in the alphabet Σ_m below.

$$\Sigma_m = \{add, change, delete, search, checkAdd, checkSearch, \\ call, checkCall, init, exit\} \cup \bigcup_{f \in Val} \{disableFeature(f)\} \cup \bigcup_{f \in Val} \{enableFeature(f)\}$$

The language of IEE IEE_{P1} is extended to allow the symbols from Σ_m, except for those symbols used to specify security concern P1, to be included in the language. The symbols used to specify security concern P1 are *add*, *search*, *change* and *delete*. L_1 is used to denote the language that represents the infinite sequences of symbols from Σ_m that does not include the *add*, *search*, *change* and *delete* symbols.

The language of IEE_{P1} is extended to allow all arbitrary interleavings of the current language IEE IEE_{P1} specifies with L_1. To do this the shuffle operator defined in definition 3.9 in section 3.4.1 is used. The operator shuffles the language of IEE_{P1} with the language L_1. The primed IEE is defined below.

$$IEE_{P1'} = (\Sigma_m, (add + search)^\omega \,||_l\, L_1) \\ where \;\; L_1 = ((checkAdd + checkSearch + checkCall + call + init + exit) + \\ \bigoplus_{f \in Val} \{disableFeature(f)\} + \bigoplus_{f \in Val} \{enableFeature(f)\})^\omega$$

The languages of IEEs IEE_{P2}, IEE_{P3} and IEE_{P45} cannot be extended in a similar way. This is because each check for an operation must directly precede the operation itself, and the shuffle operator would allow other symbols in between the check and the operation itself. Instead, the languages of IEEs IEE_{P2}, IEE_{P3} and IEE_{P45} are all extended to allow sequences of an operation preceded by its check and any of the symbols from Σ_m that does not include the operation or its check. Symbols need to be omitted to ensure the security concern is still captured by the primed IEE. The symbols omitted are *add* and *checkAdd* for IEE $IEE_{P2'}$, *search* and *checkSearch* for IEE $IEE_{P3'}$ and *call* and *checkCall* for IEE $IEE_{P45'}$. These primed IEEs are below.

$$IEE_{P2'} = (\Sigma_m, (checkAdd\ add + S_2)^\omega)$$
where
$$S_2 = ((change + delete + search + checkSearch + checkCall + call + init + exit) + \bigoplus_{f \in Val} \{disableFeature(f)\} + \bigoplus_{f \in Val} \{enableFeature(f)\})$$

$$IEE_{P3'} = (\Sigma_m, (checkSearch\ search + S_3)^\omega)$$
where
$$L_3 = ((change + delete + add + checkAdd + checkCall + call + init + exit) + \bigoplus_{f \in Val} \{disableFeature(f)\} + \bigoplus_{f \in Val} \{enableFeature(f)\})$$

$$IEE_{P45'} = (\Sigma_m, (enableFeature(tel)\ checkCall\ call\ disableFeature(tel) + S_{45})^\omega)$$
where
$$S_{45} = ((change + delete + add + checkAdd + search + checkSearch + init + exit) + \bigoplus_{f \in Val} \{disableFeature(f)\} + \bigoplus_{f \in Val} \{enableFeature(f)\})$$

The languages of IEEs IEE_{P6} and IEE_{P7} are both extended in a similar way. The language of IEE $IEE_{P6'}$ extends IEE_{P6} to allow any finite sequence of symbols from a subset of Σ_m after the message and telephone features have been disabled. Allowing a

sequence of symbols only after this ensures that at the start of the application, the message and telephone features are disabled.

The finite sequence of symbols allowed after the disabling of the message and telephone features represents the applications operations, that is, adding, calling and searching. The symbols included in this sequence are all the symbols from the alphabet Σ except for all symbols for enabling and disabling features (except for the message and telephone features) and the *init* symbol. The symbols for enabling and disabling features are omitted because security concern P6 specifies only the message and telephone features are disabled. Therefore, the enabling and disabling of other features cannot be allowed in the language. The primed IEE is below.

$$
\begin{aligned}
&IEE_{P6'} = (\Sigma, (init\ ((disableFeature(msg)\ disableFeature(tel))+ \\
&(disableFeature(tel)\ disableFeature(msg)) + (S_6)^*)^\omega) \\
&where \\
&S_6 = (add + checkAdd + search + checkSearch + call + checkCall + exit+ \\
&disableFeature(tel) + enableFeature(tel) + disableFeature(msg) + enableFeature(msg))
\end{aligned}
$$

Similarly, the language of IEE $IEE_{P7'}$ extends IEE_{P7} to allow any sequence of symbols from a subset of Σ_m before the message and telephone features have been re-enabled and the exit of the application. Allowing a sequence of symbols only before the re-enabling of the exit symbol and the message and telephone features ensures that the message and telephone features are re-enabled before the application ends. This ensures security concern P7 is still captured. The sequence of symbols allowed includes all symbols from the alphabet Σ_m except the *exit* symbol. This is shown below in the IEE $IEE_{P7'}$.

$$
\begin{aligned}
&IEE_{P7'} = (\Sigma, (S_7)^* + ((enableFeature(msg)\ enableFeature(tel))+ \\
&(enableFeature(tel)\ enablefeature(msg)))\ exit)^\omega) \\
&where \\
&S_7 = (add + checkAdd + search + checkSearch + call + checkCall + init+ \\
&\bigoplus_{f \in Val} \{disableFeature(f)\} + \bigoplus_{f \in Val} \{enableFeature(f)\})
\end{aligned}
$$

Given the six primed IEEs above, the Millionaire security policy is now defined. The Millionaire security policy is all the IEEs linked together using the *BOTH* operator. This specifies that all security concerns must be satisfied by a given program. The IEE IEE_{mil} is used to denote the combination of all the individual security concern IEEs. This is defined below.

$$IEE_{mil} = BOTH(IEE_{P1'}, BOTH(IEE_{P2'}, BOTH(IEE_{P3'},$$
$$BOTH(IEE_{P45'}, BOTH(IEE_{P6'}, IEE_{P7'})))))$$

Alternatively, this IEE could be specified as a single IEE and not as a combination of multiple IEEs. An IEE that represents all the security concerns in one would be (Σ_m, L) where L is defined below.

$$L = (init\ ((disableFeature(msg)\ disableFeature(tel))+$$
$$(disableFeature(tel)\ disableFeature(msg)))$$
$$(checkAdd\ add + checkSearch\ search+$$
$$enableFeature(tel)\ checkCall\ call\ disableFeature(tel))^*$$
$$((enableFeature(msg)\ enableFeature(tel))+$$
$$(enableFeature(tel)\ enablefeature(msg)))$$
$$exit)^\omega$$

The IEE specifies a language that supports all seven security concerns. The IEE allows sequences of an initialisation followed by the disabling of the message and telephone features in any order. This is followed by any number of add, search or call operations in any order with an appropriate check before each operation, as well as the enabling and disabling of the telephone feature before and after a call respectively. Finally, a sequence ends with the enabling of the message and telephone features in any order followed by the exit of the operation.

Both this IEE and IEE IEE_{mil} represent the same security concerns for the Millionaire application. The framework allows either representation to be used. Specifying each secu-

rity concern as an individual IEE and combining them all is much more appealing because changes in one security concern won't affect changes in any other IEE.

Given a specific program, the security concerns can be transformed into specific syntax so a specific verification tool can determine if the program satisfies the security concerns. The next section describes a specific implementation of the Millionaire game. Given this implementation, a number of configurations are defined for transforming the IEE IEE_{mil} above so verification tools can determine if the program satisfies the IEE.

4.3.5 Implementation of the *Millionaire* Application

The *Millionaire* application is implemented in Java using the MIDlet suite [FZ01]. This allows the application to be run within the Java Virtual Machine on a mobile phone.

To support the extra features provided by the application, two additional packages are implemented: the *pim* package [Sun00b] for supporting an address book, and the *telephony* package [Sun00a] for making a phone call. Only the relevant classes and methods from these packages are implemented for the case study.

In addition, a *Phone* class was created that mimics the functionality of a mobile phone. The implementation supports features for making phone calls and adding details into, and deleting and retrieving details from, the address book. The *Phone* class is implemented for verification purposes.

Currently there is no support for enabling, disabling or altering hardware features, such as the SMS or telephone features of a mobile phone. To support this for verification purposes, a simple API is implemented that allows these features, identified by port numbers, to be enabled and disabled.

The next section describes a number of configurations for transforming the Millionaire security policy into tool specific representations for the Java program.

4.3.6 Configurations

This section describes the configurations used to transform the security concerns defined in section 4.3.4. A configuration, which is a syntactic transformation, induces an IEE which induces semantics. Using syntactical transformations means the framework is light-weight and flexible. This allows ease of specifying configurations and ease of combining them together.

Instead of defining a single configuration from the implementation independent security concerns to the tool level, a number of configurations are used to gradually map the security concerns for the given Java program. This allows any changes to the policy to be more easily maintained. Furthermore, given a different implementation of the Millionaire application, some configurations can be reused.

Four configurations are defined for this case study. The first two configurations refine the abstract symbols used to specify the security concerns. At this level, the security requirements are still program independent. The last two configurations map the security concerns for the given Java program described in section 4.3.5 so specific verification tools can verify the program.

The first configuration refines the *add*, *change*, *delete*, *search*, *call*, *checkAdd*, *checkSearch* and *checkCall* symbols. The *add*, *change* and *delete* operations with their respective checks are all refined to include the user's details that the operation is adding, changing or deleting. The *search* and *call* operations with their respective checks, on the other hand, are refined to include the user's name that is going to be searched or called. The configuration t_{map} below defines this mapping. In this configuration, the value d denotes a user's details while the value n denotes a user's name. All symbols not specified in the table are mapped to themselves.

t_{map} :

add	$\bigcup_{d \in Val} \{add(d)\}$
$change$	$\bigcup_{d \in Val} \{change(d)\}$
$delete$	$\bigcup_{d \in Val} \{delete(d)\}$
$search$	$\bigcup_{n \in Val} \{search(n)\}$
$call$	$\bigcup_{n \in Val} \{call(n)\}$
$checkAdd\ add$	$\bigcup_{d \in Val} \{checkAdd(d)\ add(d)\}$
$checkSearch\ search$	$\bigcup_{n \in Val} \{checkSearch(n)\ search(n)\}$
$checkCall\ call$	$\bigcup_{n \in Val} \{checkCall(n)\ call(n)\}$

The second configuration t_{init} below maps the abstract symbols $checkAdd(d)$, $checkSearch(n)$ and $checkCall(n)$ into specific checks specified in section 4.3.3. The $checkAdd(d)$ symbol is refined into a check over a value that checks that the current number of user's added is less than two. For this check, the details of a user are not relevant and therefore are removed from the check. The value *added* is used to keep track of the number of users added into the system. This map is depicted below.

$$addCheck(d) \mapsto check(added < 2)$$

When a user's details are added, the *added* value needs to be incremented. The $add(d)$ operation is mapped into a sequence that increments the *added* value after an $add(d)$ operation.

$$add(d) \mapsto add(d)\ incr(added)$$

The $checkSearch(n)$ symbol is refined into a *check* that checks that the name searched

is one of the friends added into the system. For this, the details of an added user need to be stored. The $add(d)$ symbol is further mapped into a sequence that includes an operation for storing the users added into the system. The symbol $store(d)$ represents this. The $stored(d)$ symbol represents a function for checking if a user's details were stored in the system.

$$add(d) \mapsto add(d)\ incr(added)\ store(d)$$
$$searchCheck(n) \mapsto check(stored(n))$$

The $checkCall(n)$ symbol is refined into a *check* that checks that no more than one call is made and that the number called is one of the stored friends. The value *called* is used to represent the number of phone calls made. The $call(n)$ symbol is also refined to a sequence of operations that increments the number of calls made after a call operation. That is,

$$checkCall(n) \mapsto check(stored(n)\&\&called < 1)$$
$$call(n) \mapsto call(n)\ incr(called)$$

For the add and call checks, two values were introduced: *added* and *called*. These values need to be initialised at the beginning of the program. The symbols $init(added)$ and $init(called)$ are used to represent the initialisation of the *added* and *called* values respectively.

The methods for storing a user's details and checking if a user's details have been stored also need to be created. This allows the functions to be independent of any given program. At this level of abstraction, the symbol *storage* is used to represent all the initialisations and methods needed for storing details and checking stored details. The *init* symbol is mapped into a sequence of symbols for initialising the *added* and *called* values and for creating the *storage* methods.

$$init \mapsto init(added)\ init(called)\ storage$$

The configuration t_{init} specifying the maps specified above is below.

t_{init} :

$\forall\, d \in Val \bullet add(d)$	$add(d)\ incr(added)\ store(d)$
$\forall\, d \in Val \bullet checkAdd(d)$	$check(added < 2)$
$\forall\, n \in Val \bullet checkSearch(n)$	$check(stored(n))$
$\forall\, n \in Val \bullet checkCall(n)$	$check(stored(n) \&\& called < 1)\ incr(called)$
$init$	$init(added)\ init(called)\ storage$

The configurations t_{map} and t_{init} transform an IEE into one that is still program independent. The variables *added* and *called*, as well as the symbols $store(d)$ and $stored(n)$, are program independent. Given a C or Java program, for example, specific syntactic configurations can be specified to transform the security requirements for the given program.

This case study describes configurations for transforming an IEE for the Java program described in section 4.3.5. The next section defines a configuration for mapping the IEE into Java specific syntax.

Configurations for Java Syntax

Until now, the configurations have mapped symbols to symbols, where symbols include placeholders. This allows the symbols to still be both language and program independent. A configuration can now be specified mapping the symbols into syntax of a specific language, for example Java or C. In this case study, the configurations for mapping these symbols into Java syntax are shown.

The next configuration, t_{syn}, maps the checks, increments and initialisations into Java specific syntax. The checks, increments and initialisations introduce the variables *added* and *called* into a given program. These variables are created and used independently of any given program. This ensures these variables are not altered by the program. Therefore,

regardless of the Java program being verified, these mappings won't need to be changed.

All increments have the same syntax. Therefore, a function similar to t_{assert} can be used to map symbols of the form *incr(v)*, where v is the variable being incremented into the symbol $v = v + 1;$. The function f_{incr} below defines this where $I = \{added, called\}$

$$f_{incr} : \boxed{\forall\, i \in I \bullet incr(i) \mid i = i + 1;}$$

A similar function can be defined for variable initalisations.

$$f_{init} : \boxed{\forall\, i \in I \bullet init(i) \mid int\ i = 0;}$$

Both the functions f_{incr} and f_{init} are Java specific maps. For different programming languages, these functions need to be changed. However, for different Java implementations of the Millionaire application, these functions don't need to be changed.

The specific implementation of the application is now considered. The next function maps symbols into symbols specific for the program being verified. All symbols not mentioned are mapped to their identity. If the implementation of the application uses different symbols for adding, search, etc, these mappings then need to be defined.

The abstract symbol *storage* is used to represent all the necessary initialisations and methods for storing a user's details and checking if a specified name has been stored by the user. For these methods, the format used in the implementation for storing a user's name and details needs to be known. This means the function may need to be changed for each program to depict the format used to represent these values in the implementation.

To store a user and check if a user has been stored, two methods are created and inserted into a given program: *stored* and *store*. The function *stored* returns true if a user's details have been stored given their name and returns false otherwise. The function *store* takes as an argument a user's details and stores it into a data structure. The data structure used in

this case study is an array. By inserting these methods and data structure into a program it ensures a program does not edit these security relevant details.

The *storage* symbol is further mapped into two initialisations for storing details in an array and keeping track of how many details have been stored. It is assumed there is some class *Detail* in the program that stores a user's details and has appropriate methods for accessing a user's name and number. It should be noted that if a program does not contain this class, by default it will be rejected. The symbol *storage* is mapped into the code below.

```
private Detail details[2] = new Detail();
private int numDetails = 0;

public void store(Detaild){
    details[numDetails] = d;
    numDetails++;
}
```

```
public boolean stored(String name){
    for(int x = 0; x < numDetails; x++){
        if(details[x].getName.equal(name))
            return true;
    }
    return false;
}
```

The function f_{store} specifies a mapping from the symbol *storage* into the code above. For user brevity, the function is omitted. Finally, the configuration t_{syn} below defines the mappings of the three functions described above.

$$t_{syn} = f_{store} \circ_c f_{incr} \circ_c f_{init}$$

At this stage, a number of configurations can be used to map the requirements into a tool specific representation. For this case study, two Java verification tools are used: JPF and Bandera. Each of the tools JPF and Bandera should be able to verify all of the given security concerns in the Java program. Therefore, the final configuration t_{tools} specifies that the Java program satisfies the initial security concerns if it is verified by either of the verification tools. The configuration makes use of the configurations t_{assert} and $t_{bandera}$ previously defined for the JPF and Bandera tools respectively.

$$t_{tools}(C) = EITHER_c(t_{assert} \circ_c C, t_{bandera} \circ_c C)$$

The configuration t_{tools} takes an argument, C. This configuration specifies that the configurations C are used to transform an IEE first followed by t_{assert} to generate one IEE and similarly again for $t_{bandera}$ to generate a second IEE. The resulting IEEs are combined together using the *EITHER* operator specifying that either of the tools JPF or Bandera must satisfy their respective programs for the Java program to satisfy the initial security concerns.

4.3.7 Program Verification

The four configurations defined could be applied to each of the individual IEEs representing each of the security concerns. This would allow a program to be verified one property at a time. The flexibility of the framework allows the same configurations to be applied to the combination of all the security concerns, that is, IEE IEE_{mil}. For this case study the latter method is chosen. Using this method allows all the security concerns to be verified at once.

The IEE IEE_{mil} representing all the security concerns was mapped using the sequence of configurations below.

$$t_{tools}(t_{syn} \circ_c t_{init} \circ_c t_{map}\ IEE_{mil})$$

The end result of the transformations is two IEEs: one in the syntax for the JPF verification tool and the other in the syntax for the Bandera tool. Figure 4.5 shows a pictorial view of the transformations.

Given the two IEEs, the assert statements, initalisations and incrementations were automatically inserted into the Java program. Given an instrumented program, JPF was able to verify the respective instrumented program while the Bandera tool didn't accept its respective instrumented program. Since satisfaction was specified to only rely on either of

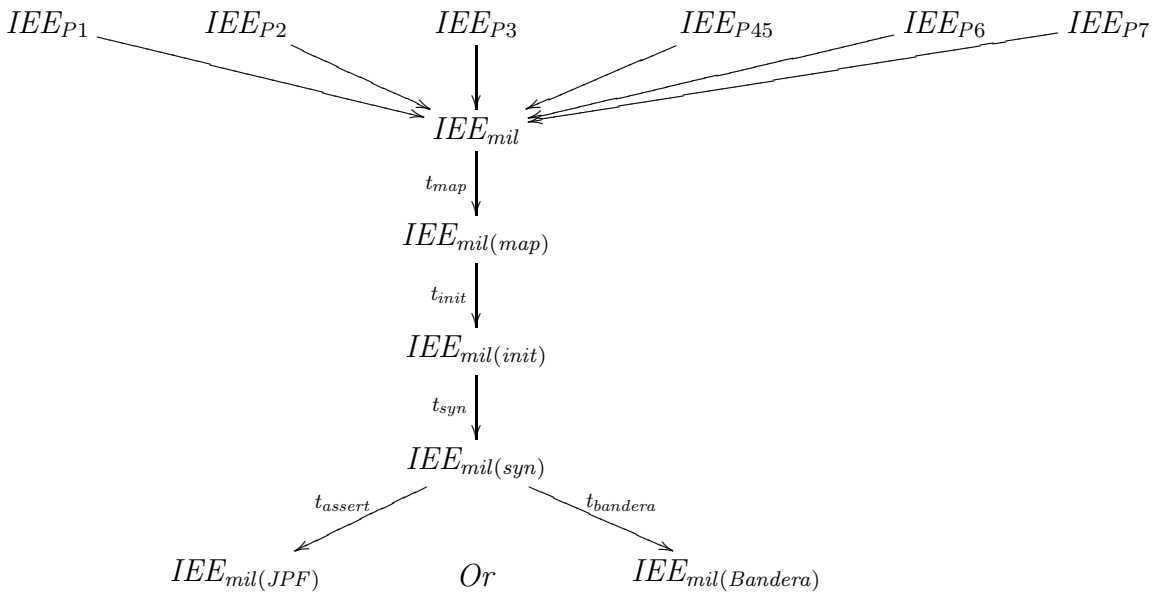

Figure 4.5: Path of Transformations taken for the Millionaire Policy

the verification tools to verify the program, the Java program satisfied the initial security concerns due to JPF verifying its respective instrumented program.

4.3.8 Discussion

This case study demonstrates that a real world example program can be verified, given a user's security requirements, using the framework presented in this dissertation.

In this case study, a user's security requirements are initially specified independently of the implementation of the application. Each security concern at this initial level is specified as a single IEE and linked together using the *BOTH* operator. Specifying each security concern as a single IEE allows changes in a security concern to only affect their corresponding IEE.

Given the initial IEE representing the security concerns, a number of configurations are used to transform that IEE into representations for the JPF and Bandera verification tools. The first two configurations refine the abstract symbols in the initial IEE. The next configuration maps the symbols into Java specific syntax with the last configuration

mapping the symbols for the JPF and Bandera verification tools. The end result of the transformation process is the two annotated Java programs for the JPF and Bandera tools.

The flexibility of the framework presented in this dissertation allows a user's security concerns to be represented in multiple ways. The security concerns could be represented with each concern in its own IEE combined together to form one IEE, or with all the security concerns represented directly in a single IEE. Regardless of which method is chosen, the same configurations specified in this case study can be applied to any of the initial IEEs.

The Java program in this case study satisfies the initial user's security requirements if either tool JPF or Bandera verified their respective annotated program. While in theory both tools should be able to verify the properties specified on the *Millionaire* application, in practice it proved challenging with only one tool being able to verify the Java program in the end.

One of the main problems faced was attempting to verify the application within the MIDlet suite. This was solved by creating stubs for all of the classes and methods used from the MIDlet suite. Functionality remained the same (to the extent it needed to), however, source files now exist and the overall size of the stubs was drastically smaller than the MIDlet suite. Size of an application is also a common problem in verification [ZLL04, Zit03].

After creating stubs for the MIDlet suite, further alterations need to be made to the program to remove the IO system used. Since IO is undecidable, JPF was unable to verify applications with IO support. To solve this, the *random* function supported by JPF [Hav99] was used. The *random* function is used in switch statements to determine which case in the switch statement is executed. When verifying a switch statement, JPF chooses all paths as possible executions of an application. This removes the undecidability of user input without changing the functionality of the application.

After slightly altering the code to remove all user input, and replacing the condition

statements dependent on user input with JPF's *random* function, JPF was able to accept the application for verification. Initially, the verification process never terminated. After analysis of the application, it was established that the verification process entered an infinite loop. This was caused by the application's design. The application supported a user friendly interface in which players could, in certain menus, go back to the previous menu. This caused JPF to go into a continuous loop when verifying the application. Once all of the back options in the application were removed, JPF successfully verified the application.

Application design also lead to Bandera being unable to accept the instrumented *Millionaire* application for verification. Bandera rejected the application because of the existence of recursion in the application.

The flexibility of the framework allows ease of specifying configurations. This is particularly useful for inducing multiple tool representations as this case study showed. Having the ease in specifying multiple configurations that allows redundant tool representations to be induced is useful in a case such as this, where the application design meant one of the tools didn't accept the program. However, since another tool was able to verify the program, it can be verified that the program satisfied the initial user's security requirements. Thus, the framework as a whole was successful in this case study.

Comparison to the Hets Framework

This section describes how this case study could be encoded in the Hets framework and the similarities and differences between the Hets framework and the framework used in this case study.

Using the Hets framework, a user could describe the Millionaire security concerns in some logic. Similar to how the security concerns were initially specified for this case study, the user could describe the security concerns independently of any implementation.

The user could then encode a specific implementation of the Millionaire application in a similar or different logic. Proof obligations would then need to be specified saying that the implementation is a refinement of the abstract security concerns. That is, that the implementation adheres to the security concerns. The system would be sound if the proof obligations could be proved by a theorem prover.

In both the Hets framework and the framework used in this case study, the security concerns are represented at multiple levels of abstraction. At one level the security concerns are represented independently of any specific implementation, while at another level, they're for a specific implementation.

In the Hets framework, the language used to describe each level of abstraction needs to have a precise syntax and semantics defined in the framework. As an example from this case study, the abstract symbols *check*, *checkAdd*, *checkSearch* and *checkCall* for example would need to have a semantics defined for each symbol. In the framework presented in this dissertation, these checks are syntactically mapped to JPF and Bandera assertions. If a C program was implemented, these symbols could be mapped into Blast assertions, for example. The semantics of these checks are induced by the verification tools used to determine program satisfaction.

This case study demonstrates the abilities of the framework presented in this dissertation. The flexibility of the framework allows the security concerns arising from the Millionaire application to be represented in a number of ways. Regardless of the method chosen, a number of configurations are created allowing the security concerns to be transformed into tool specific representations for the JPF and Bandera verification tools. Given the successful verification of JPF for the Java program implemented for this case study, it is successfully verified that the Java program adheres to the Millionaire security concerns.

Chapter 5

Discussion

Ensuring the security of programs is an important issue. It may not always be the case that a single verification tool is capable of verifying if all program behaviours of a program satisfy a given policy. This dissertation describes and presents the implementation of a framework for using multiple existing verification tools to verify a program.

In the framework, a user's security requirements are represented as intermediate execution environments. IEEs provide a mechanism for allowing a user's security requirements to be split into multiple representations linked via appropriate operators. Using configurations to define how to map from one IEE to another, an IEE can be automatically transformed through a series of transformations into multiple tool specific representations.

To validate the framework three case studies have been performed and presented in the previous chapter. The *memory* case study demonstrates how a single abstract policy can be mapped into multiple representations. Depending on the implementation of memory in a program and the verification tool used to verify the program, a different sequence of configurations are required. Furthermore, some configurations are designed such that they can be reused.

The *bank* case study demonstrates how concurrency can be handled in the framework.

In the case study, a synchronized block is used to ensure thread non-interference for the Java *bank* program. The case study used a number of intermediate levels to transform the initial IEE. This means that if a specifier wanted to change the mechanism used for ensuring thread non-interference, depending on the new mechanism used, only the mechanism level would need to be changed.

The *millionaire* case study demonstrates the framework in the larger picture. The initial IEE is substantially larger than the other case studies. This is due to the complexity of the policy requirements. The flexibility of the framework allows the security policy to be represented in a number of ways. Regardless of the method chosen to represent the security policy, a number configurations were created to transform the security policy into tool specific representations for the JPF and Bandera verification tools. Given a different implementation of the Millionaire application, different configurations can be plugged into the framework allowing the respective tool representations to be induced from these configurations.

Using intermediate levels, each initial policy is gradually mapped through a sequence of configurations into a number of IEEs for specific verification tools. Using intermediate levels allows each configuration to be clearly specified. This allows the policy to be maintained more easily as a change at the abstract level can be followed through to the tool level(s). Furthermore, individual configurations, specifically the configurations for mapping a policy into a specific tool can be reused.

5.1 Conclusion

Limited work has been done on combining policy languages. The aim of this dissertation is to describe a framework for leveraging verification tools to enhance the verification technologies available for policy enforcement. It was initially stated that the aims of this

dissertation were:

1. Using the concepts viewed in existing policy languages, to use language theory to capture a user's security requirements for multiple representations.

2. Using language theory, to build a framework for transforming a user's security requirements from an initial representation into multiple representations so multiple verification tools could be used.

3. To aid in the transformation process, use manually created configurations to describe how to transform between representations and how to link these transformed representations.

4. To define what it means for a program to satisfy a user's requirements in terms of the generated representations of those requirements and the tools used to enforce those requirements.

These aims have been achieved by providing the following contributions that have been detailed in previous chapters:

1. To capture a user's security requirements, IEEs are used. IEEs allow a user's security requirements to be represented at multiple levels of abstraction.

2. A framework was designed and implemented that allowed IEEs to be transformed between different levels of abstraction. The framework allows a user's security requirements to be automatically transformed into multiple representations for multiple verification tools.

3. To define how one IEE is transformed into another, the framework uses configurations which are manually defined by the user.

4. The semantics of the operators used to link IEEs in the framework is defined so it can be induced if a given program satisfies a user's security requirements given the results of multiple verification tools.

The main limitation of the framework presented in this dissertation is that it is assumed that configurations are appropriately defined. This is a result of the configurations being purely syntactical. In the framework it is possible to specify a configuration that maps some security property to true, which may allow an unsafe program to be determined safe. Hence, the assumption is that the creator of the configurations will encode them correctly, according to the original security requirements.

The framework presented in this dissertation contributes to the field by providing a new approach allowing multiple verification tools to be used to determine program satisfaction that is both light-weight and flexible.

5.2 Future Work

In the Millionaire case study, there was substantial work required in manually changing the Java code so the program would be accepted by the Java PathFinder tool. An area of future work would involve creating configurations to specifying how to map a program so that it could be automatically mapped along with an IEE.

Currently, if multiple tools are used to perform verification, specifically using the *BOTH*and *IMP*operators to link languages, each tool is individually run on the given IEE and the results are manually looked at to determine whether the overall verification passed or failed. Automating the running of the verification tools and automating some desirable output to the user is another area of future work.

Finally, mitigating the limitation mentioned in the previous section is also an area of future work.

Bibliography

[AB01] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Lecture Notes in Computer Science*, 2030:25–41, 2001.

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[ABLP92] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–23, London, UK, 1992. Springer-Verlag.

[AG97] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AHM89] A. Avron, F. Honsell, and I. Mason. An Overview of the Edinburgh Logical Framework. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification*, pages 323–340. Springer-Verlag, 1989.

[BB02] Mustapha Bourahla and Mohamed Benmohamed. Predicate Abstraction and Refinement for Model Checking VHDL State Machines. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.

[BDC+95] J. M. Bradshaw, S. Dutfield, B. Carpenter, R. Jeffers, and T. Robinson. KAoS: A Generic Agent Architecture for Aerospace Applications. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.

[BFK99] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures (Position Paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[BFS98] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *Financial Cryptography*, pages 254–274, 1998.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:505–525, 2007.

[Bib87] W. Bibel. *Automated Theorem Proving*. Friedrich Vieweg & Sohn, Braunschweig, Second edition edition, 1987.

[Bis03] Matthew Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[BO05] Piero Bonatti and Daniel Olmedilla. Driving and Monitoring Provisional Trust Negotiation with Metapolicies. In *POLICY '05: Proceedings of the Sixth IEEE*

International Workshop on Policies for Distributed Systems and Networks, pages 14–23. IEEE Computer Society, 2005.

[BR02] T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. *In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 1–3, 2002.

[BS03] P. A. Bonatti and P. Samarati. Logics for Authorization and Security. In *Logics for Emerging Applications of Databases*, pages 277–323, 2003.

[CC03] S. N. Chari and P. Cheng. BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[CCG+03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proceedings of the 25th International Conference on Software engineering*, pages 385–395. IEEE Computer Society, May 2003.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[CEE+01] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[CFL+97] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29:953–964, 1997.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, December 1999.

[CLN00] C. Colby, P. Lee, and G. C. Necula". Proof-Carrying Code Architecture for Java. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 557–560, London, UK, 2000. Springer-Verlag.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX security symposium*, pages 63–78, San Antonio, Texas, Jan 1998.

[CW02] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, 2002.

[DDEL01] N. Damianou, N. Dulay, and M. Sloman E. Lupu. The ponder policy specification language. In *Proceedings of Policy*, number 1995 in LNCS, pages 18–39, 2001.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, 1999.

[Eli98] J. Elien. Certificate Discovery Using SPKI/SDSI 2.0 Certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[FGMP90] A. Felty, E. Gunter, D. Miller, and F. Pfenning. Tutorial on Lambda Prolog. In *Proceedings of the 10th International Conference on Automated Deduction*, page 682, London, UK, 1990. Springer-Verlag.

[FQ02] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 191–202. ACM Press, 2002.

[FZ01] Y. Feng and Dr. J. Zhu. *Wireless Java™ Programming with J2ME*. Sams Publishing, 2001.

[GEP+95] R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[GJ02] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. IEEE Computer Security Foundations Workshop*, pages 77–91. IEEE Press, 2002. Full version appeared in J. Computer Security 12(3/4).

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the JavaTM Development Kit 1.2. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[Gon98] L. Gong. Java Security Architecture (JDK1.2). Technical report, 1998.

[Hav99] Klaus Havelund. Java Pathfinder User Guide. http://ase.arc.nasa.goc/havelund, August 1999.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

[HK00] J. Howell and D. Kotz. A Formal Semantics for SPKI. Technical Report TR2000-363, Dartmouth College, Computer Science, Hanover, NH, March 2000.

[HMHX07] V. C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance Checking of Access Control Policies Specified in XACML. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 275–280. IEEE Computer Society, 2007.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP00] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[Jim01] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In IEEE Press, editor, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, USA, 1997.

[Kag02] L. Kagal. Rei : A Policy Language for the Me-Centric Project. Technical report, HP Labs, September 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-270.html.

[KFJ03] L. Kagal, T. Finin, and A. Joshi. A Policy Language for a Pervasive Computing Environment. In *In IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 63–76, 2003.

[KHPC01] D. R. Kuhn, V. C. Hu, W. T. Polk, and S. Chang. Introduction to Public Key Technology and the Federal PKI Infrastructure, 2001. National Institute of Standards and Technology.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[Lam74] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[Li00] N. Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization.* PhD thesis, New York University, 2000.

[LS01a] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42. The USENIX Association, June 2001.

[LS01b] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enchanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development Graphs – Proof Management for Structured Specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142. ACM Press, 1997.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, pages 186–197, 1998.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, Hets. In *TACAS*, pages 519–522, 2007.

[Mos05] T. Mossakowski. Heterogeneous Theories and the Heterogeneous Tool Set. In *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germ, 2005.

[Mye99] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM Press, 1999.

[Nec97] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, France, Jan 1997.

[NL96] G. C. Necula and P. Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, September 1996.

[NL98] G. C. Necula and P. Lee. Efficient Representation and Validation of Proofs. In *Proceedings of the 13th Annual symposium on Logic in Computer Science*, pages 93–104. IEEE Computer Society, 1998.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth Symposium on Logic Programming*, pages 810–827, 1988.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Jun 1992. Springer-Verlag.

[Pfe89] F. Pfenning. Elf: A Language for Logic Definition and Verified Meta-Programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, 1989. IEEE Computer Society Press.

[Pfe91] F. Pfenning. Logic Programming in the LF Logical Framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe96] F. Pfenning. The Practice of Logical Frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, 1996. Springer-Verlag LNCS 1059.

[PS99] F. Pfenning and C. Schurmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206. Springer-Verlag, 1999.

[PS05] V. Patil and R. K. Shyamasundar. Trust Management for E-Transactions. In *Sadhana: Academy Proceedings in Engineering Sciences*, volume 30, pages 141–158. Indian Academy of Sciences, April/June 2005.

[RL96] R. L. Rivest and B. W. Lampson. SDSI – A Simple Distributed Security Infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[RW89] E. Ruf and D. Weise. Nondeterminism and Unification in LogScheme: Integrating Logic and Functional Programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 327–339. ACM Press, 1989.

[SF01] S. Smalley and T. Fraser. A security policy configuration for the security-enhanced linux. *NAI Labs Technical Report*, February 2001.

[SM03] A. Sabelfeld and A. Myers. Language-based information flow security. *IEEE Journal on Selected Aread in Communications*, 21(1):5–19, 2003.

[Sma02] S. Smalley. Configuring the selinux policy. NAI Labs Report #02-007, June 2002.

[SRRS01] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-Carrying Code (MCC): A New Paradigm for Mobile-Code Security. In *NSPW*

'01: Proceedings of the 2001 workshop on New security paradigms, pages 23–30, New York, NY, USA, 2001. ACM Press.

[Sta02] Frank Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, 2002.

[Sun00a] Sun Developer Network (SDN). Java Telephony API (JTAPI). http://java.sun.com/products/jtapi/, March 2000.

[Sun00b] Sun Developer Network (SDN). JavaPhone Specification 1.0. http://java.sun.com/products/javaphone/index.jsp, March 2000.

[SVB+03] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2003. ACM Press.

[SVS01] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. NAI Labs Report #01-043, December 2001.

[TLMM07] P. Torrini, C. Lueth, C. Maeder, and T Mossakowski. Translating Haskell to Isabelle. Technical report, 2007.

[Val98] A. Valmari. The State Explosion Problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - Second Generation of a Java Model Checker. In *Workshop on Advances in Verification*, 2000.

[VPS02] V. N. Venkatakrishnan, Ram Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 61–68. ACM Press, 2002.

[WL92] T. Y. C. Woo and S. S Lam. Authorization in distributed systems : A formal approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 33–51, 1992.

[WL93] T. Y. C. Woo and S. S. Lam. A framework for distributed authorization. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 112–118, New York, NY, USA, 1993. ACM Press.

[Zda03] S. Zdancewic. A type system for robust declassification. In *Nineteenth Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science*, 2003.

[Zit03] M. Zitser. Securing Software: An Evaluation of Static Source Code Analyzers. *Master's thesis, Massachusetts Institute of Technology*, 2003.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.